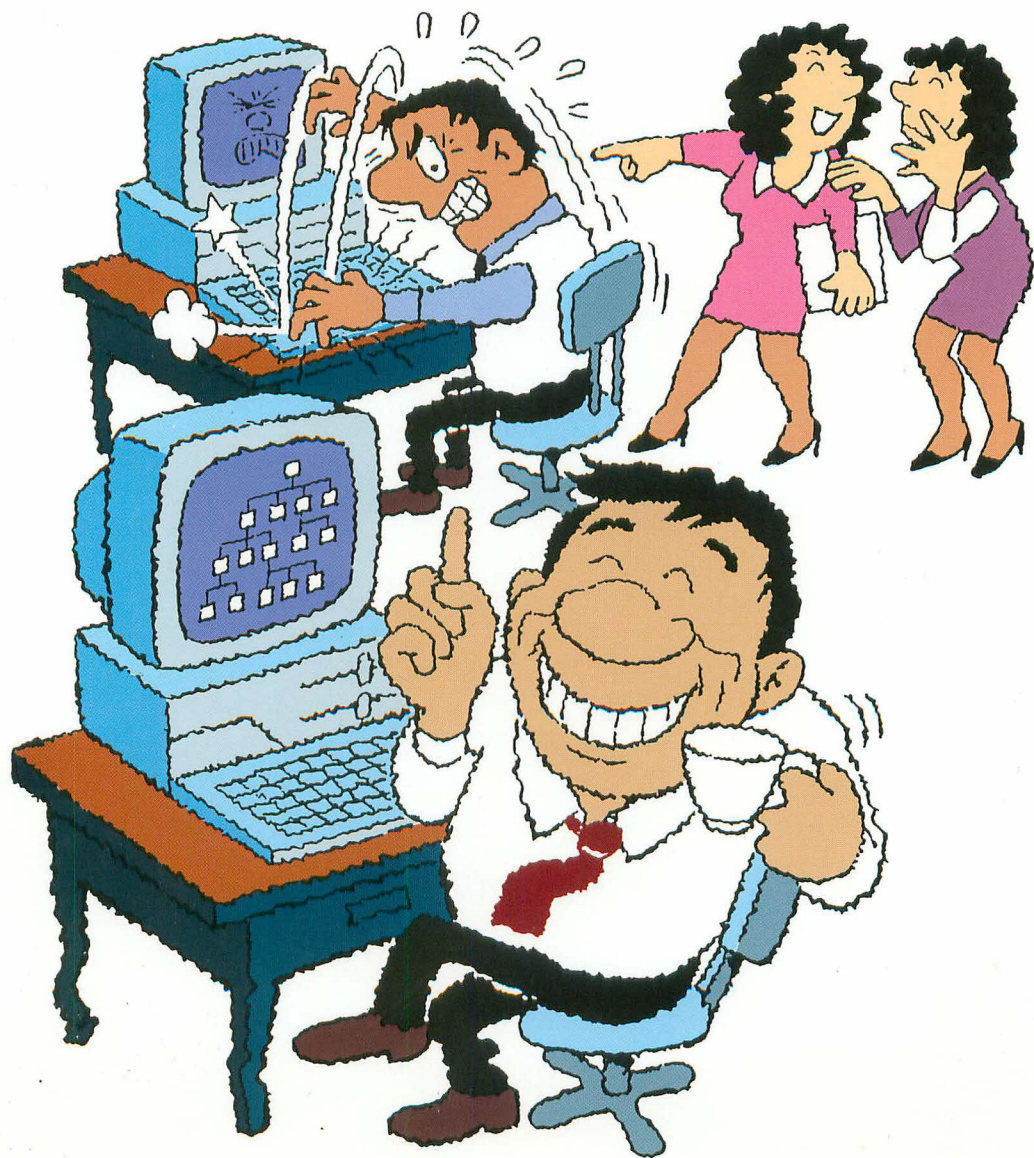
Cによるらくらく構造化設計

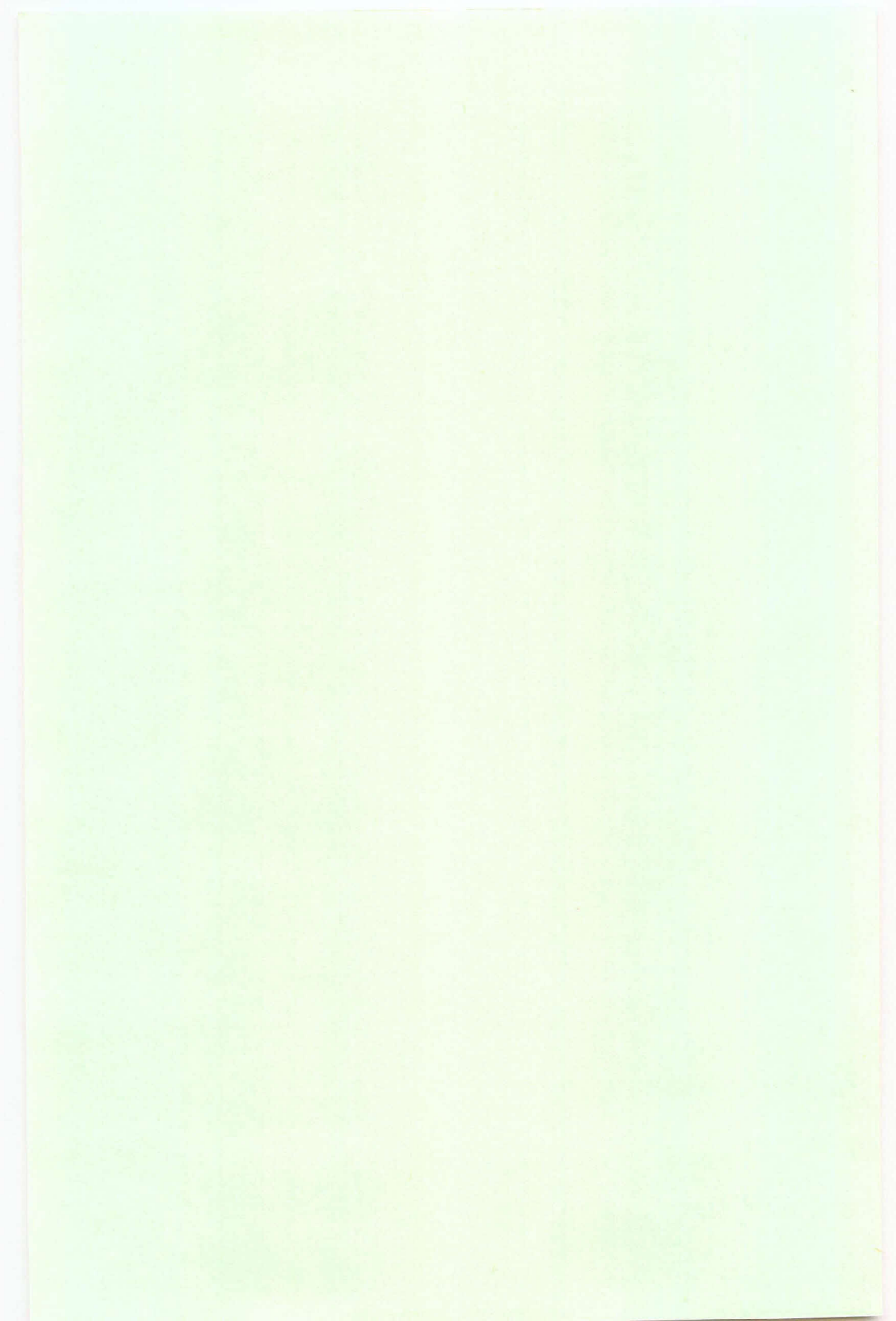
FINE SOFT series

ソフト開発の効率アップ!

國友義久 著

《本書の内容》

●いまではパソコン・ソフトといえども，大型機やワークステーションとかわらないぐらい大規模な開発が行われています．

●大きなソフトウェアを効率よく，高品質につくりあげるために考えられたのが「構造化設計」です．

●永年この分野の教育に携わってきた筆者が，構造化のコツをやさしく解説します．

Cによるらくらく構造化設計

ソフト開発の効率アップ!

國友義久 著

FINE SOFT series

CQ出版社

はしがき

ソフトウェアの信頼性は，一般にハードウェアの信頼性より低いといわれています．しかし，テストにかける費用は，ソフトウェアのほうがハードウェアより数段多いのが普通です．つまり，ソフトウェアは手間をかけているにもかかわらず，製品としての信頼度はいまひとつといったところなのです．

これはソフトウェアの技術者が，ハードウェアの技術者より腕が劣っていることを意味するのでしょうか．そうではないと思います．基本的な問題は，ソフトウェアとハードウェアの複雑さのちがいにあります．

ハードウェアは，形式と機能がきまっている機械命令を解釈し，忠実に実行すれば事がすみます．しかし，ソフトウェアはそうはいきません．入力は適用業務ごとに異なりますし，処理の仕方も業務ごとに異なります．また，機械命令はそれ自体個々に独立していて，他の命令間とのインターフェースをとくに考える必要がないのに対し，ソフトウェアで処理する問題には，機能間に複雑な関連があるのが普通であり．そのインターフェース設計は大変やっかいです．

このように，ソフトウェアの信頼性の低さは，その複雑さに起因します．だからといって，信頼性の低さを容認してよいわけではありません．ゲーム・ソフトのエラーは，まだ笑ってすますことができるでしょうが，工場の生産ラインを制御するソフトウェアのエラーは，企業に大きな損害を与えるでしょうし，ばあいによっては，人命にかかわるような問題になることもあります．

それだけに，信頼性の高いソフトウェアを作成しなければならないという必要性は大きいのです．しかし，現実には，ソフトウェア作成は，まだまだ個人に依存する部分が多く，信頼性の問題も，まだまだ個人の技術に依存しがちです．個人の技術への依存度を低めた高信頼性ソフトウェアの設計手法の確立が望まれています．

本書は，その１つの考え方である構造化(モジュール化)設計について，特にマイコンでプログラムを作成している方を対象に，詳しく紹介しています．

構造化設計の考え方は，もともと大型のコンピュータでプログラムを作成するときに，有効な手法として注目されてきました．しかし，最近のように，マイコンといえども大き

な記憶容量をもつようになると，マイコンで処理するプログラムも，従来の大型コンピュータで扱っていたのと同等の規模のものも出現するようになりました．

したがって，構造化設計の必要性は，マイコンの分野まで広がってきています．エレクトロニクス技術の進歩は，今後，マイコンの処理能力をますます高めていくことになるでしょう．それとともに，プログラムの構造化設計の重要性も大きくなってきます．

すでにふれたように，本書はマイコンでプログラム開発を行う方を対象に，できるだけわかりやすくプログラムの構造化設計を説明することを目的としています．そのために，まず内容的にできるだけ具体性をもたせるよう努めました．文章を平易にすることはもちろん，例題，図表をできるだけ数多く用いました．内容的には，設計部分は特定のプログラミング言語に依存するものでなく，すべての問題に汎用的に適応できるものです．ただコーディング例は，マイコン技術者間で広く普及しているC言語，FORTRANをベースにしました．

本書は全部で9章から構成されています．第1章から第3章までは，プログラムの構造化設計を行うさいに身につけておかなければならない基本的事項について説明しています．まず，この部分をしっかり理解してください．第4章から第6章までは，構造化設計を具体的に展開していくときの手順，考慮点を中心に述べています．構造化設計の中心をなす部分ですので，説明にも大きなスペースをさいてあります．

第7章は，分割した個々のモジュールの論理の設計について説明しています．コーディングに直接つながっていく重要な部分であり，プログラムの読みやすさはこの設計のよしあしにかかっています．

第8章では，コーディングしたモジュールをテストしながら，1つのプログラムにまとめていく方法について説明しています．最後の第9章では，構造化設計をより実際的に身につけるために，本格的な例題をもとに，設計からコーディングまでを一貫して説明しています．

本書全体を通して読むことにより，プログラムを構造的に設計し，コーディングまでもつていく方法を具体的に理解していただけるはずです．

最後に，本書の出版にご尽力くださいましたCQ出版社の山本 潔さん，大野 浩さんならびに，㈱エディックスの中山洋一さんに心からお礼申しあげます．

1991年　12月

著　者

CONTENTS

CONTENTS

CONTENTS

〈カバー・デザイン〉マサ・クリエイティブ 〈カバーおよび本文イラスト〉榎本幸一郎

1．プログラム開発の現状を理解しよう

コンピュータがハードウェアだけなら，何もできない単なる箱であり，そこに命をふき込むにはソフトウェアが必要であることは，いまならたいていの人は知っています．それだけに，ソフトウェア，すなわち，プログラムの開発はコンピュータの出現と同時に行なわれてきたわけですが，プログラムを作るという作業は，なにぶん，人手に頼る部分が多く，そこからいろいろな問題が発生しています．

そこで，まず最初に，いままでのプログラム開発で何が大きな問題であったか，それを振返ってみることからはじめましょう．それらの問題を正しく認識し，その解決策を考えることが，とりもなおさず，効果的なプログラム開発方法を考えることにもなるわけです．

1.1　いままでのプログラム開発の問題点は何だろう

同じ仕様のプログラムを10人のプログラマにあたえ，プログラムを作らせたら，10個の異なったプログラムができあがると言われることがよくあります．それくらいプログラムを作る仕事は属人性が強く，個人の技能に頼っているのが現状です．

本書をお読みになっている読者の方々は，おそらく，いままでに何らかのプログラミングの経験をお持ちの方が多いと思いますが，もしあなたがプログラム仕様書を渡され，プログラムを作ってほしいと頼まれたなら，まずどんな作業から手をつけていくのでしょうか．

仕様書の内容を理解したら，早速コーディング用紙に向かい，最初の命令からコーディングにとりかかる方が，たぶん，たくさんいらっしゃると思います．あるいは，コーディング用紙ではなく，直接コンピュータ画面とにらめっこしながら，キーボードから手ぎわよく命令を入力していく人も多いことでしょう．

腕に自信のある方は，他の誰よりもよいプログラムを早く作成できる力を自分は持って

いるとひそかに思っておられるかも知れません(図1.1).

その自信に水をさすつもりはないのですが，自分の作ったプログラムに対して，下記のような観点から，検討したとき，どれだけ自信をもって答えられるでしょうか.

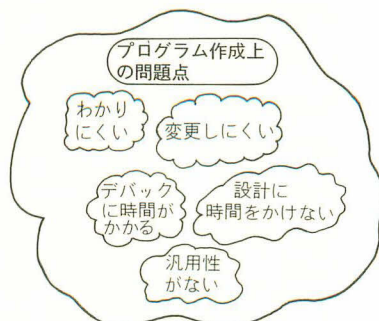
- 誰がみてもわかりやすいプログラムになっているだろうか
- あとで仕様の変更が発生したとき，変更しやすいプログラムになっているだろうか
- プログラム作成時に，エラーの修正にてまひまかけなかっただろうか
- プログラムの設計という問題にどれだけ真剣に取組んだろうか．そもそも，プログラムの設計とは何だろうか．設計に対して明確な基準をもっていただろうか
- できあがったプログラムの一部は，あとで他のプログラムを作成するとき，再利用できるだろうか

これらの問題にすべて自信をもってイエスと答え，しかも，その理由を明確に説明できる方はプログラマとして大変優秀な方だと思います．しかし，現実には，イエスと答えられる人よりも，首をかしげてしまう人の方がずっと多いでしょう.

実は，これらの問題は，従来のプログラム開発の問題点として指摘されてきた重要なものばかりなのです(図1.2).

図1.1　よいプログラマとは

3人とも自信満々．でも…

図1.2 プログラム作成上の問題点

他人より少しでも命令数の少ないプログラムを書き，実行時間を早くする．これこそがプログラマとしての腕の見せどころと考えておられる方がまだまだ多いのではないでしょうか．確かに，この考え方は一理あるのですが，できあがったプログラムを先の5つの観点からみたときどうでしょうか．

おそらく，命令数の少ない，実行の速いプログラムは，その論理が大変複雑になり，お世辞にもわかりやすいとは言えないはずです．開発時に，いろいろな試行錯誤が行われ，それだけ犯したエラーの数も増えてくる．また，変更しようとすれば，その変更の波及効果は大きく，プログラム全体を書き直さざるを得ない．もちろん，プログラムの一部を他に流用することなどとても不可能である．…まあ，こういったところに落ち着くのではないかと思います．

これらのことをいくつかの簡単な例でみていくことにしましょう．

1.1.1　問題点のあるプログラム例

```
        if(p>0)
        goto nq;
        if(w>0)
        goto ns;
        t=5;
        goto nr;
ns:     s=4;
        goto nr;
nq:     if(q>0)
        goto nx;
        y=2;
        if(v<=0)
        goto nr;
        z=3;
        goto nr;
nx:     x=1;
        if(v<=0)
        goto nr;
        z=3;
nr:     r=6;
```

▶特定の命令を実行するための条件さがし

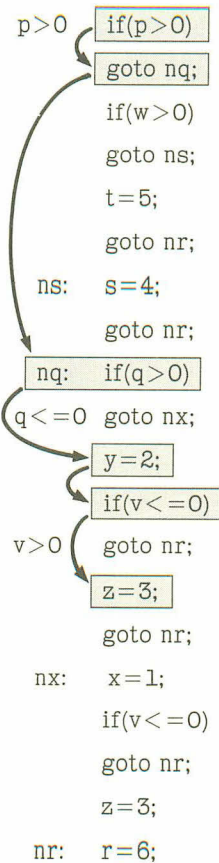
左のコーディング例は，C言語で書かれたプログラムの一部ですが，ここで

z=3

の命令を実施するのはどのような条件のときかを考えてみてください．

z=3の命令は2箇所にあらわれていますので，少し慎重に読み取っていく必要があります．まず，pの値が正かどうかをif命令でテストしています．pが正であるときは，goto命令でnqという名札のところへ飛んでいきます．

一方，pが正でないときは，つぎのテスト，すなわち，wが正であるかどうかを調べます．そして，wが正であれば，名札nsへ飛びます．wが正でないときは，t=5の命令を実行し，その後，名札nrへ飛びます．wが正のとき，名札nsへ飛んだ後は，s=4の命令を実行し，その後，さらにnrへ飛んでいきます．

さて，最初の条件テスト，pが正であるかどうかを調べて，pが正であったときは，nqへ飛びました．nqでは，つぎのテスト，すなわち，qが正であるかどうかを調べます．qが正であれば，名札nxへ飛びます．正でないときは，y=2の命令を実行後，さらに別のテスト，vが正でないかどうかを調べます．vが正でないときは，名札nrへ飛びます．そして，vが正のときはz=3の命令が実行されます．やっと目的の命令に到着できましたね．ここまでの経路をプログラム上でたどると次のようになります．

z＝3の命令がどのような条件のときに実行されるのか，いままでの追跡を整理することで確かめてみてください．まず，vが正であるときであることはいま確かめたばかりですからすぐわかります．しかし，vのテストを実施するのは，どのような条件のときかを調べる必要があります．

それは，qが正でないときでしたね．また，qのテストはどのようなときにテストされたでしょうか．それは，pのテストで，pが正であったときでした．

ここで，やっと答に到着できました．答は，

- pが正である
- qが正でない
- vが正である

の三つの条件が満足されたときです．

だいぶ，つかれましたね．しかし，残念ながら，これで問題が解決したわけではありません．

z=3の命令は，このプログラムでは，もう1箇所，別の箇所で実行されています．すなわち，名札nxの部分です．ここでのz=3の命令が実行される条件も調べる必要があります．せっかく，ここまできたのでですから，めんどうでももう少しがまんして調べてみてください．ここでは，答だけあげておきますので，皆さんの調べた結果と較べてみて正しいかどうかを確めてください．

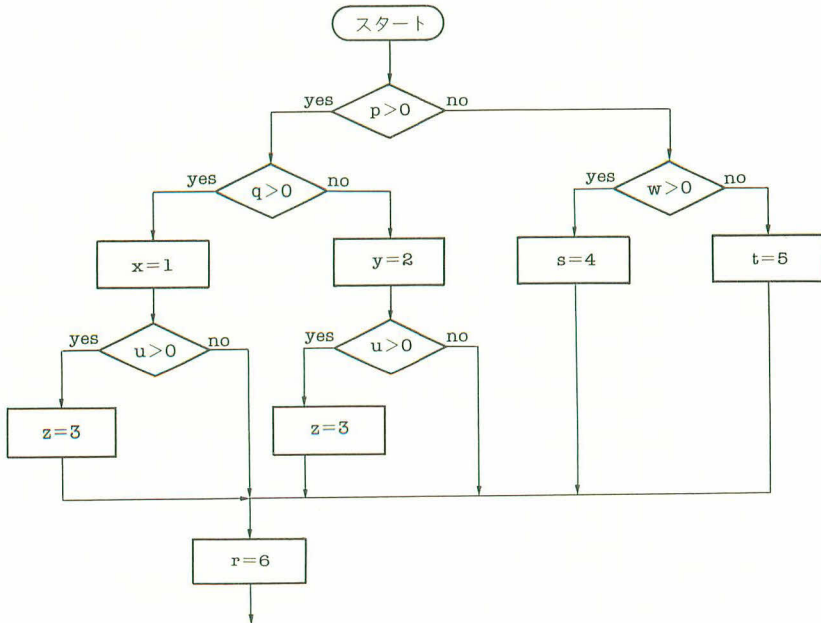
- pが正である
- qが正である
- vが正である

先に出した答と比較してみてください．少しちがいますね．そうです．「qが正でない」が，こんどは「qが正である」になっています(他の二つの条件は同じ)．

この二つの答を合わせて考えれば，q は正であってもなくても，z=3 は実行されることになります．したがって，q の条件ははずしてかまわないことになります．そこで，求める答は

- p が正である
- v が正である

の二つの条件が満足されたときとなります．このプログラムの論理をそのまま流れ図であらわすと図1.3 のようになります．この流れ図でみれば，コーディング例よりもかなりはっきりと z=3 が実行される条件を追いかけることができますね．

図1.3　悪いコーディング例の流れ図

▶このプログラムの特徴

それにしても，だいぶ手間どりましたね．このコーディング例の特徴はなんでしょうか．それは，goto 命令がたくさん使われていることです．それとともに，名札も多く使われています．このことが，このプログラムを必要以上にわかりにくくしていることは，すでに読者の方は気が付かれていることと思います．

▶プログラム例の改良

このわかりにくいプログラムを，もっとわかりやすいものにするにはどうしたらよいのでしょうか．わかりにくさが goto 命令の乱発に起因しているのですから，goto 命令をできるだけ使わないようにすれば，わかりやすくなるはずです．

そこで，先ほどのプログラムの論理を変えないで，goto 命令を使わないプログラムを作ってみましょう．結果を下記に示します．

```
if(p>0)
     {
      if(q>0)
        x=1;
      else
        y=2;
      if(v>0)
        z=3;
     }
else
     {
      if(w>0)
        s=4;
      else
        t=5;
     }
r=6;
```

このコーディング例では，z=3の命令は1箇所にしか出ていません．その分，考えやすいですね．しかも，goto命令が一つも使われていませんので，先ほどのように，あっちへ飛んだり，こっちへ飛んだりすることなくプログラムを追跡できます．さらに，if命令のレベルに応じて，命令の書き出し桁をずらして書いてあります(字下げのルールと呼ぶ)ので，その結果，ifの制御範囲を大変明確に把握することができます．

z=3の命令は，pの条件テストが真のときで，かつvの条件が真のとき実行されることが，視覚的にも大変よくわかるはずです．

同じ論理をコーディングしたにもかかわらず，先ほどの例よりずっとわかりやすくなっていることを理解できたと思います．

このわかりやすいプログラムは，構造化プログラミングという技法を用いることで作ることができます．構造化プログラミングについては，**コラムA**に簡単にまとめましたが，詳細は後の章であらためて検討することにします．

このように，プログラムをわかりやすく作成すれば，当然のことながら，エラーの発生頻度は少なくなります．また，たとえエラーをおかしたとしても，それをみつけ，修正する時間は少なくてすむことになります．

いままでの統計によれば，エラーの検出と修正に費す時間は，プログラム作成に費す全体時間の3割から5割を占めるといわれています．この時間が減少するということは，そのままプログラミングの生産性向上に大きく寄与することになります．

▶変更，追加のしやすさ

また，プログラムをわかりやすく作っておけば，後で，そのプログラムに変更，追加の必要ができたとき，大変やりやすくなります．

たとえば，このプログラム例で，

- p>0
- v<=0

のとき，u=6の命令を追加実行させるように変更したくなったとします．

▶わかりにくいコーディングへの追加

最初にあげたわかりにくいコーディング例ではどうなるでしょうか．先程と同じようにこのコーディングの論理を追跡していき，u=6の命令を追加する場所をさがさなくてはなりません．その詳細説明はくりかえしになるので省きますが，結構大変な作業でしたね．

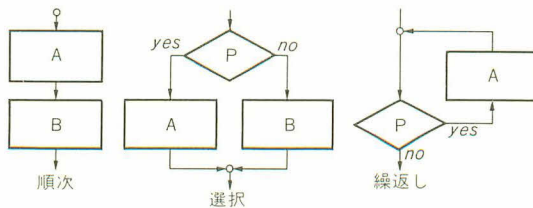
コラムA

ストラクチャード・プログラミング

構造化プログラミングとも呼ばれています．イタリアのコンピュータ技術者，ベームとヤコピニが最初に理論的に証明して注目されました．適正プログラム(入口が一つで出口が一つのプログラム)であれば，その内容の如何にかかわらず，順次，選択，繰返し(図A)の3つの基本的制御構造の組合せによって，プログラムの論理を作成できるという構造化定理を用いてプログラミングするものです．

その後，オランダのアイントホーヘン大学のダイクストラ教授によって世界的に紹介され，プログラミングがアートから科学になったと評価されました．

実用的にはそれまでの難解なプログラムをわかりやすいプログラムへ転換できる技法として注目され，現在多くのプログラミングで採用されています．

図A　3つの基本的制御構造

結果は下記の示すように，矢印の処，2箇所に追加することになります．めんどうでも，自分でしっかり確めてみてください．

```
        if(p>0)
        goto nq;
        if(w>0)
        goto ns;
        t=5;
        goto nr;
ns:     s=4;
        goto nx;
nq:     if(q>0)
        goto nx;
        y=2;
        if(v<=0)
        u=6;        ←追加
        goto nr;
        z=3;
        goto nr;
nx:     x=1;
        if(v<=0)
        u=6;        ←追加
        goto nr;
        z=3;
nr:     r=6;
```

▶わかりやすいコーディングへの追加

一方，わかりやすいコーディング例ではどうなるでしょか．こちらはgoto命令はありませんし，ifの制限範囲も大変明確ですから，すぐ追加箇所はみつけられますね．pの条件が真のときの実行部分で，vのテストが偽の部分を追加すればよいのです．結果はつぎのようになります．追加しても，わかりやすさは少しもそこなわれていない点に注目してください．

```
if(p>0)
        {
         if(q>0)
           x=1;
         else
           y=2;
         if(v>0)
           z=3;
         else
           u=6;        ←追加
        }
  else
        {
         if(w>0)
           s=4;
         else
           t=5;
        }
  r=6;
```

これまでにあげた例は，理解を容易にするために，ごく簡単な問題でした．したがって，わかりにくい例といっても，少し注意深く調べればミスをおかすことはまずないといってよいでしょう．わかりにくさ，わかりやすさは，あくまでも相対的なものであり，絶対的なものではありませんでした．しかし，問題そのものが複雑になってきたとき，この差は大変大きなものになります．

1.1.2　プログラムの機能設計（構造化設計）の例

さて，いくつかの例でわかりにくいプログラムとわかりやすいプログラムについて考えてきましたが，このようなことを考えるのがプログラムの設計なのでしょうか．

これまでの例は，プログラムの論理のよしあしが検討のテーマになっていました．論理のよしあしについて考えるのも立派にプログラムの設計のなかに入ります．しかし，それがすべてではありません．論理の設計は，プログラム設計の一部なのです．そして，プログラムの設計を考えるときは，論理の設計の前に，ぜひ考えておかなければならないひとつのステップがあります．それはプログラムの機能設計（構造化設計）です．機能設計と論理設計のちがいの詳細は，後であらためて検討するとして，ここでは，具体例を中心に考えてみることにします．

まず，プログラムの機能設計とは何かについて簡単にふれておきましょう．プログラムの機能設計では，最初にプログラムで解くべき問題をひとつの全体機能として把えます．たとえば，「3次元方程式を解く」，「大気汚染度を測定する」，「入院患者を監視する」といったように．その全体機能をいくつかの部分機能に分割し，それらを階層構造的に合成（複合）していきます．そして，階層構造上の各機能を個々のモジュール（正確な定義は後述）に対応させます（図1.4）．

簡単な例でみてみましょう．図1.5は，座席予約システムをごく簡単に類型化し，機能設計を行った例を示しています．

図1.4　機能設計の概略

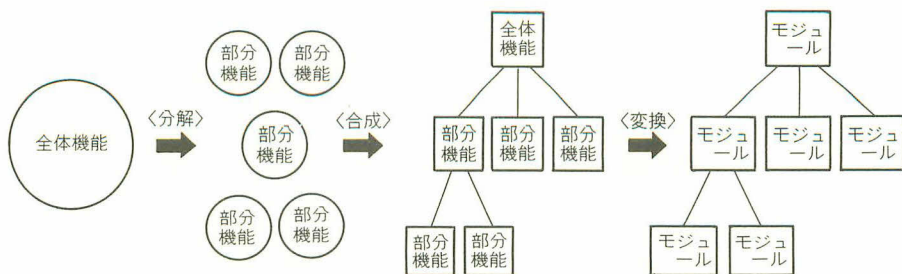
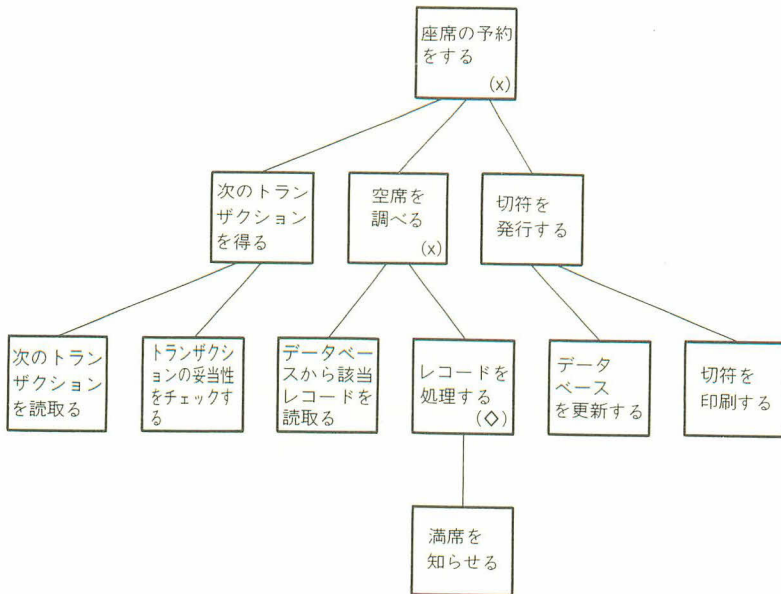



図1.5 座席予約システムの機能設計例1

▶機能分割

この問題の全体機能は「座席を予約する」です．図1.5は，この全体機能を部分機能に分割し，それらを階層構造として合成した結果を示しています．「座席を予約する」機能は，「次のトランザクションを得る」，「空席を調べる」，「切符を発行する」の三つの部分機能から構成されます．さらに，これら三つの部分機能は，図1.5に示した，より小さな部分機能に分割されます．たとえば，「次のトランザクションを得る」は，「次のトランザクションを読み取る」と「トランザクションの妥当性を調べる」の二つの部分機能から構成されます．

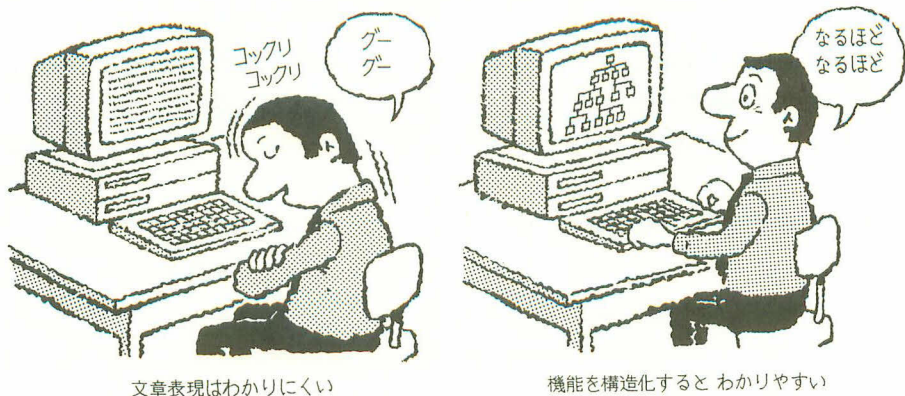
要は，階層構造の各レベルの機能は，より下位レベルの機能の総括とみることができます．このようにみてくると，「座席を予約する」という全体機能を実施するためには，どんな部分機能を実施しなければならないかが段階的に詳細化されていて，大変理解しやすくなっています．この理解のしやすさは，プログラムの仕様をつぎのように文章でだらだらと一枚岩的に記述されたものと比較すれば，よくわかっていただけるはずです．

「このプログラムは座席の予約を行うためのものである．座席の予約を行うためには，まず要求トランザクションを読み取る必要がある．要求トランザクションは，別紙仕様に示すような一定の形式をもっていなければならない．

正しいトランザクションであれば，そのトランザクションの要求していること，すなわち，何月何日の何便の予約をしたいかを知り，該当便に対するレコードをデータベースから読み取り，空席があるかどうかを調べる．空席があれば，切符を発行し，なければ，満席であることを端末操作員に知らせる．」

おそらく，現実の予約システムの仕様はこれよりもっと複雑で細部にわたる説明が延々と続けられているはずです．それを全部読み取り，正確に理解し，プログラムのコーディングまでもっていくのは大変な作業になることでしょう．プログラムのコーディングを行う前に，仕様書で述べていることを図1.5のように整理するかしないかでは，そのコーディングのやりやすさは随分とちがってくるはずです(図1.6)．論理設計の前に機能設計が必要であると述べた理由は，まさに，この点にあります．

図1.5のように機能分割したあと，個々の機能をモジュール対応させ，その後，モジュールの論理を設計していけばよいのです．その場合にも，わかりやすい論理が要求されるのは，先の例でみてきたとおりです．

図1.6　機能階層図の効果

1.1.3　プログラム設計の評価

さて，図1.5に示したような機能設計の結果，プログラムで行うべきことが大変理解しやすい形で提示されたことは認めていただけると思います．しかし，この結果は，設計として，はたしてベストなものなのでしょうか．機能設計の結果がよい設計になっているのか，何か問題点があるのかをどのような観点から評価すればよいのでしょうか．われわれは，いままで，設計結果に対する客観的な評価尺度をもっていたのでしょうか．

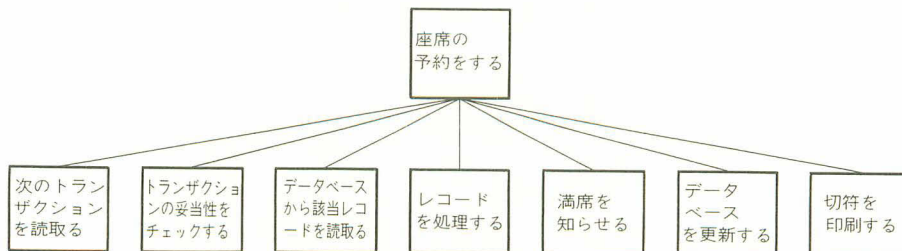
このことの詳細については，おいおい明らかにしていきますが，設計の結果に対する評価は，いろいろな側面から把えていくことができます．

図1.5の結果を例にとって，いくつかの側面から評価してみましょう．

▶情報隠匿の概念

先にも述べたように，この結果を機能的な展開図として評価すれば，なかなかよくできています．このプログラムで何を行うべきかが明確に提示されています．また，機能を階層的に展開するときに心掛けなければならない情報隠匿の概念も達成されています．情報隠匿の概念については，後の章であらためて述べますが，簡単にいえば，階層構造のそれぞれのレベルで，そのレベルでは不必要だと思われる詳細情報はかくしてしまい，そのレベルでの詳しさに設計の焦点を絞り，設計をやりやすくしようとすることです．

たとえば，「座席の予約をする」ためには，どんな機能が必要になるかを考えるとき，必要な機能を大きなものから小さなものまですべて同時に把えて考える(図1.7)のではなく，まず，機能を大まかに把えます．予約を要求している取引データ(トランザクション)を読み込み，そのトランザクションが要求している便の空席を調べ，空席があれば切符を発行するというのが座席予約の基本であることは誰でも理解できます．

図1.7　座席予約システムの機能設計例 2

したがって，座席予約システムの機能階層図の上から2番目のレベルを，図1.5に示したようにすれば，図1.7の例よりわかりやすくなるはずです．この段階では，トランザクションを読み取るときに必要な細かな機能，あるいは，空席を調べるために必要な細かな機能は特に考えていません．それによって，座席予約のためには何が必要か，その大枠をしっかり把えることができるのです．これが情報隠匿の概念であり，設計時点での大切な考え方です(図1.8)．

このような観点で，図1.5をもう一度見直していただければ，機能の詳細化が階層的にうまく行われていることがよくわかりますね．それがこの設計結果を図1.7よりわかりやすいものにしている主因です．

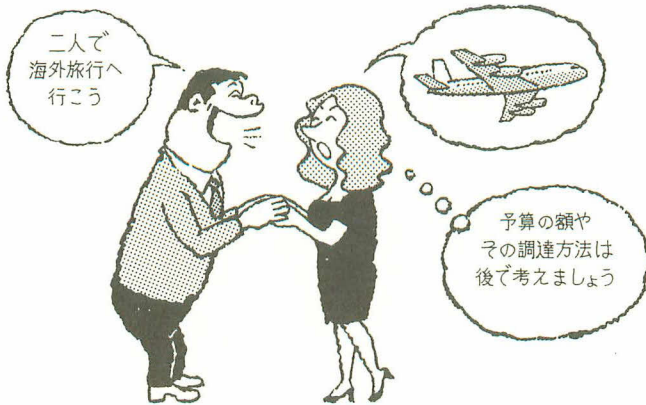
ところで，プログラムの設計作業では，まず，機能設計が必要であり，その結果得られた機能階層図の個々の機能をモジュールに対応させ，そのあと，個々のモジュールの論理を設計するのがよいことを先に簡単にふれました(図1.4)．機能構造図は，モジュール構造図に変換され，そのあと，コンピュータで稼動できるようにコーディングされ，テストされます．したがって，機能設計の結果は，コンピュータでプログラムとして稼動するときに何か問題を発生しないかどうかの観点からもみていく必要があります．

▶影響範囲と制御範囲

図1.5の結果をその面からみるとどうでしょうか．たとえば，この問題の性格から，一番中心になる機能は，おそらく，「空席を調べる」のところになることでしょう．図1.5では，空席を調べるために，まず，トランザクションが要求している便に対するレコードをデータベースから読み取ります．そして，そのレコードを調べて空席があるかどうかをチェックしています．空席があれば，切符を発行します．一方，空席がなければ，満席であることを端末操作員に知らせ，処理すべきつぎのトランザクションを読み取るようにしなければなりません．

このことから，図1.5の機能構造図の「レコードを処理する」機能には，該当の便に空席があるか否かの決定が含まれることになります．そして，この決定結果によって，プログラムが次に行うべき機能が異なってきます．

図1.5では，満席であれば，「レコードを処理する」機能の一部として「満席を知らせる」機能を実行するようになっています．しかし，満席のばあいは，それだけではなく，つぎのトランザクションを得ることも必要です．図1.5では，これは，「レコードを処理する」機能の一部としてではなく，別の機能として把えられています．そして，この機能

図1.8 情報隠匿の概念

を実行させるための制御権は，階層構造図の最上位機能「座席の予約をする」がもっています．

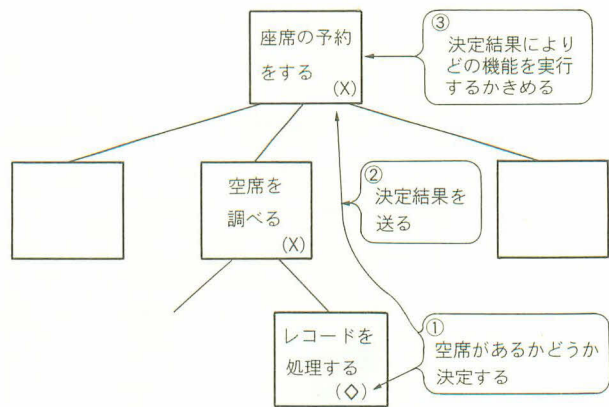
このことは，「レコードを処理する」機能のなかで行われた決定結果を「座席の予約をする」機能に教えてやる必要があることを示唆しています．そのためには，「レコードを処理する」と「座席の予約をする」の中間にある「空席を調べる」機能を介して，情報を知らせる必要があります(なぜ，そうなのかはモジュールの制御規約を理解する必要がありますが，詳細は後の章で述べます)．

したがって，「レコードを処理する」のなかの決定問題は，「空席を調べる」と「座席の予約をする」機能に何らかの影響をあたえることになります．一つの決定問題がどこまで影響をあたえるかを，その**決定問題の影響範囲**と呼びます．図1.5の例では，決定問題を(◇)，影響範囲を(×)で示してあります．

この決定問題は，空席があったことも意味しています．「切符を発行する」機能を実行する制御権は，その上位機能である「座席の予約をする」がもっています．

以上から，つぎのことがいえます．

- 「レコードを処理する」機能のなかで行われた決定の結果は，「空席を調べる」機能を介して，「座席の予約をする」機能に送られる．
- 「座席の予約をする」機能は，その決定結果を判断し，「次のトランザクションを得る」

図1.9 決定問題

機能か「切符を発行する」機能のどちらを実行するのかを決める必要がある(図1.9).

設計結果の評価として，ここで問題にすべきことは，決定がおよぼす影響範囲です．このばあい，影響範囲が大きくなればなるほど，その設計は好ましくないと言えます．

何事でもそうですが，良い影響なら，その及ぼす範囲が大きければ大きいほど好ましいのですが，悪い影響は，なるべくならその及ぼす範囲は小さくしたいものです．

ここで論じている影響範囲は，好ましくない影響ですから，できるだけ小さいほうがよいのです．「座席を予約する」機能や「空席を調べる」機能からみれば，他の機能のなかで行われた決定によって，自分が何らかの影響をうけるのは，はた迷惑な話です．自分のことだけ考えていればよいのではなく，他のことも気にしなければならないのですから(図1.10).

これは機能の独立性を低め，他の目的のために再使用する可能性をそれだけ低下させます．また，変更があったとき，その波及範囲を広め，エラーの発生確率を高めてしまうことになります．できれば，影響は自分が制御できる範囲内に留め他に影響を及ぼさないようにしたいものです．そして，そのような設計がよい設計なのです．

プログラムの機能構造図は，モジュール構造図に変換され，プログラムとして実行されます．その意味では，モジュールのなかで行なわれる決定の影響範囲は，そのモジュールの制御範囲のなかに留めるようにするべきです．

モジュールの制御範囲は，そのモジュール自身とそのモジュールが呼び出すモジュール(下位モジュール)です(図1.11).

この観点から図1.5の結果を評価すれば，影響範囲が制御範囲をはみ出していて問題が

図1.10 ほかへの影響はできるだけ少なく

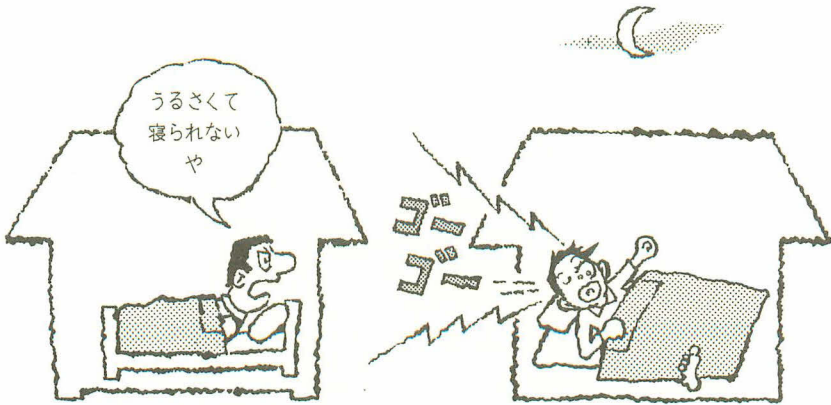
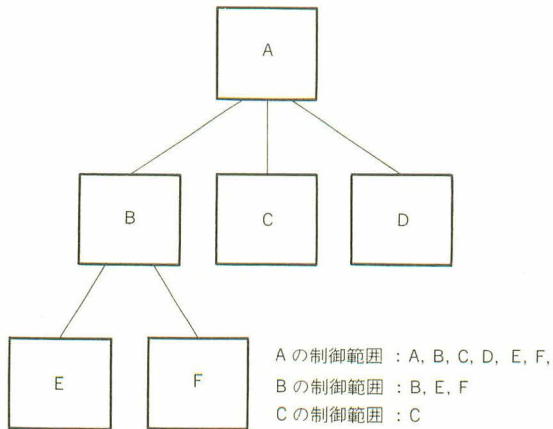
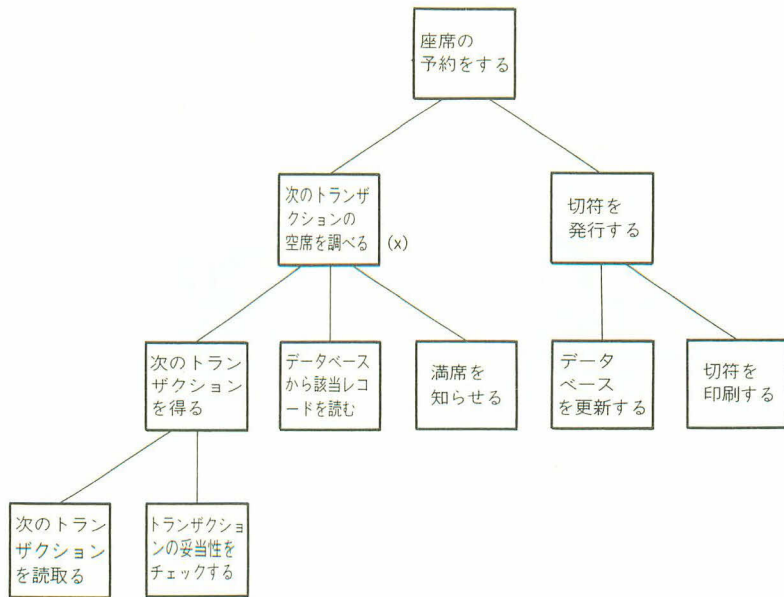



図1.11 制御範囲は自分自身と下位モジュール

あることがわかります．影響範囲を制御範囲内に収めるには，どうしたらよいでしょうか．この問題は，基本的には，設計者の創造力にゆだねられる分野です．まさに，設計者として知恵のだしどころです．

この例題に対する設計の代替案を図1.12に示しておきましょう．図1.5の結果とよく較べてみて下さい．どこがちがっていますか．

図1.12 座席予約システムの代替え設計案

そうですね．図1.5の階層構造の上位から2つ目のレベルでの3つの機能が，図1.12では，2つの機能にまとめられています．すなわち，「次のトランザクションの空席を調べる」と「切符を発行する」の2つです．

特に，「次のトランザクションの空席を調べる」機能にしたところが，この代替案のみそです．機能をこのようにまとめることにより，この機能を実行するための部分機能(下位機能)として，「次のトランザクションを得る」，「データベースから該当レコードを読む」，「満席を知らせる」の3つを設定することができます．

この案では，先に検討対象にした決定問題は，当然のことですが，「次のトランザクションの空席を調べる」機能のなかで行なわれすことになります．決定結果が，満席であれば，「満席を知らせる」機能と「次のトランザクションを得る」機能を実行します．決定結果が空席ありであれば，実行制御権を「座席を予約する」に戻します．このとき，「座席を予約する」機能は，決定結果を気にせず，「切符を発行する」機能を実行します．なぜなら，制御権が戻ってくるのは，空席があるときに限られているからです．満席のばあいは，「次のトランザクションの空席を調べる」機能の制御範囲だけで対応処理が可能であり，制御範囲外に決定の影響をもたらすことはありません．

このことからみても，図1.12の設計結果が，図1.5の結果より好ましいと評価してよいでしょう．図1.12の結果は，先の情報隠匿の概念をも満たしています．

▶共通機能の分離

それでは，図1.12の結果は設計としてベストのものなのでしょうか．問題の内容をつぶさに分析していけば，まだこの結果にも改良の余地があることに気がつきます．

たとえば，「トランザクションの妥当性をチェックする」機能があります．この機能のなかでひとつの決定問題が行なわれることでしょう．すなわち，トランザクションの形式を調べた結果，それが妥当であるか否かの決定です．このとき，妥当でなければ，トランザクション・エラーを何らかの形で知らせなければなりません．図1.12の設計では，このことが「トランザクションの妥当性をチェックする」機能のなかに包含されています．

一方，「データベースから該当レコードを読み取る」機能にも注目してみましょう．ここでは，トランザクションや要求した便に対する該当レコードの読取り作業が行なわれます．しかし，このさい，何らかの理由で該当レコードがデータベース上にない可能性を考えておく必要があります．たとえば，トランザクションが形式的に妥当であっても，その内容の一部で指定した便が実在しないものである可能性があります．

このようなときに，該当レコードなしのエラー・メッセージを打出す必要があります．

そうです．いまとりあげた2つの機能，「トランザクションの妥当性をチェックする」と「データベースから該当レコードを読取る」は，ともに，「エラー・メッセージを打出す」機能を包含しているのです．しかも，「エラー・メッセージを打出す」機能は，エラ

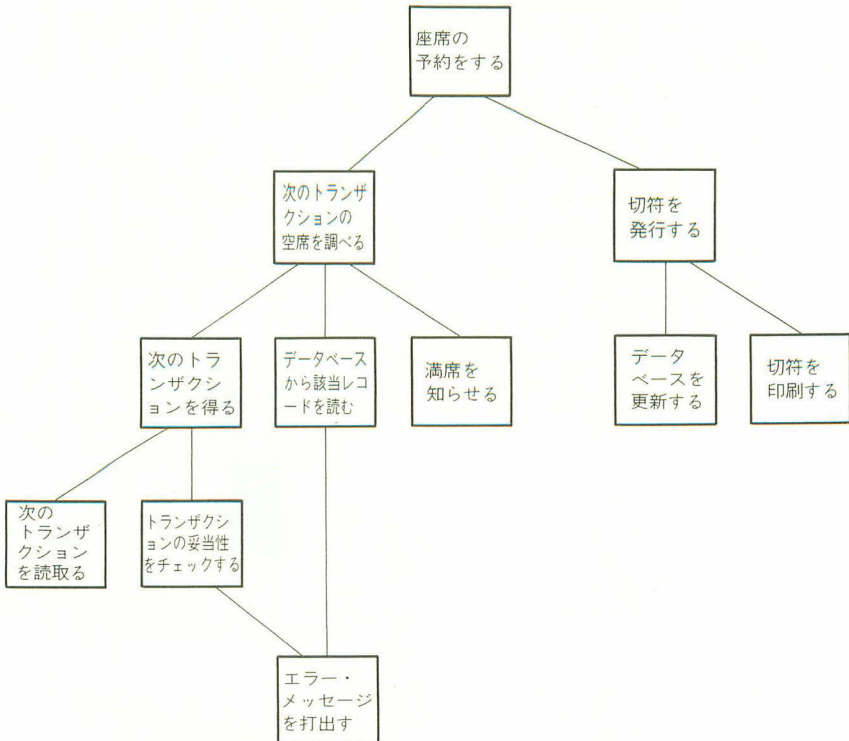
図1.13 同じことを繰り返しても意味がない

ー・メッセージの内容は異なりますが，打出す機能自体は2つの機能間で共通であることでしょう.

これは，この2つの機能をモジュールとしてコーディングするさいに，同じものを重複してもつことを意味しています．そのようなことになれば，作成の手間も重複することで大きくなりますし，プログラムもそれだけ大きくなり，無駄なコンピュータ資源を必要とします(図1.13)．それだけではなく，将来，エラー・メッセージ打出し機能に変更が生じたとき，変更個所が複数になり，手間もかかるし，ばあいによっては，エラーの原因にもなります.

このような問題を避けるには，複数個所での共通機能は，それだけで独立させ，必要に応じて，その機能を呼び出すようにプログラムを設計すればよいのです.

この観点から，図1.12の結果は，図1.14のように修正した方がよいことがわかります.

図1.14 座席予約システムの最終設計案

▶まとめ

このように，ひとつの設計問題でも，それをいろいろな側面から評価することにより，よりよい品質の設計に近づけていくことができます．

重要なのは，設計評価の尺度をしっかりと設定することです．尺度がなければ，その設計結果がよいのか，よくないのか評価のしようがありません．評価したとしても，それは評価者の主観的な色彩が強くなり，客観的なものではありえないでしょう．

いままでプログラム作成では，このあたりに大きな問題点があったのは先に指摘したとおりです．極論すれば，いままでのプログラム作成では，正しい意味でのプログラム設計は行われていなかったといっても過言ではありません．いままで述べたような，機能設計を行なわずに，プログラムの仕様書から直接コーディングを行なうのが常ではなかったかと思います．そこでは，プログラムの設計とは，論理の設計だけを意味していました．

そのようにして作成されたプログラムは，モジュール化も不十分で，ひとつのプログラムのなかに，多くの機能がざつぜんとしてならべられ，時間がたてば，自分が作ったプログラムでも，何が何だかわからないような難解なものになりがちでした．そのようなプログラムは，当然，変更を困難にし，エラーの発生頻度が大きくなり，他の目的のために再利用する可能性などなきに等しいといった代物になってしまうことでしょう．

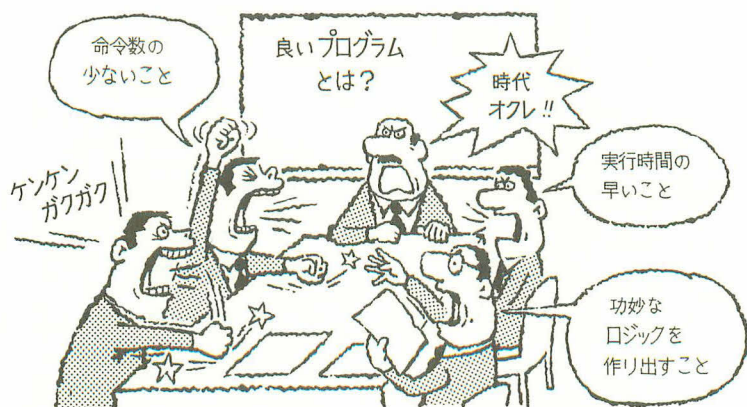
こういったことになってしまう最大の原因は，くりかえすようですが，プログラムの設計に対する配慮が不足していることにあるのです．

それでは，プログラムの設計とは何かについて整理してみましょう．この命題について考える場合，まず第1に，はっきりさせておかなければならないことがあります．

それは，良いプログラムとは何だろうかということです(図1.15)．なぜなら，プログラムの設計は，良いプログラムを作るためのものだからです．

良いプログラムとは，ひとことで言えば，わかりやすいプログラムのことです．いままでのプログラムは難しく作る傾向がありました．それは，たくみな論理によって，命令数を少なくし，実行時間も速いプログラムが良いプログラムと考えられていたからです．このようなプログラムは，どうしても難しいプログラムになりがちです．難しいプログラムは，作るときにエラーをともなうことが多くなるし，その修正に多くの時間を必要とします．変更に対処しにくいし，再利用の可能性も少なくなることは，すでに指摘したとおりです．

したがって，良いプログラムは，わかりやすいプログラムということになるのです．そして，プログラム設計の目的は，いかにわかりやすいプログラムを作るかにあります．本

図1.15 「よいプログラム」とは

書の目的も，まさに，ここにあります．わかりやすいプログラムを作成するために，プログラムをどのように設計していくか．いままで，いくつかの具体例をとおして，機能設計(構造化設計)と論理設計の必要性を説いてきました．

このことについて，こんどはもう少し体系的に説明することにしましょう．そのためには，まず，プログラムの開発手順を整理し，そのなかでのプログラム設計の位置づけと役割を正しく理解する必要があります．

1.2 プログラム開発の手順を整理してみよう

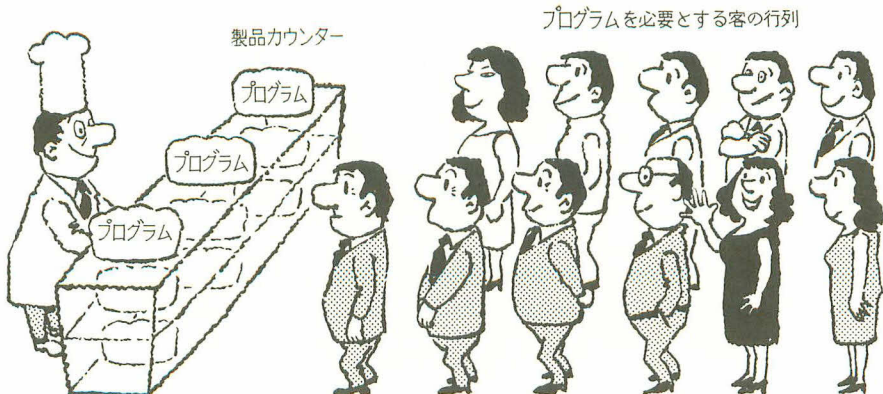
プログラムは1つの製品です．コンピュータでプログラムを稼動させることにより，ある種の問題の解決をはかります．パン焼き機はパンを焼くためにあります．洗たく機は衣服を洗たくするためにあります．それぞれ特定の機能をもっています．

製品としてどんな機能をもつべきかは，その製品の利用者の要求によって決まります．プログラムも他の製品と同様です．プログラムにどんな機能をもたせるべきかは，そのプログラムの利用者がプログラムに何を要求しているかによってきまります．

したがって，プログラム開発は，そのプログラムに対する要件を定義することから始まります．

製品に対する要件が定義できれば，次はその要件を満たすために，製品をどのように設計するかを考えなければなりません．パン焼き器をどのように設計すれば他社製品よりもおいしいパンが焼けるようになるのでしょうか．家庭の主婦が買いたくなる洗たく機はどのように設計すればよいのでしょうか．製品としての価値は，設計のやり方いかんできまります．

プログラムに関しても，この点はそのまま当てはまります．プログラムの製品としての価値は，その設計いかんによってきまります．利用者によろこんで使ってもらえるようなプログラムとなるように設計していかなければなりません(図1.16)．

図1.16 利用者のほしがるプログラムをつくる

このように，1つの製品を開発していくためには，一定の手順が存在します．プログラムを1つの製品とみなしたとき，プログラムを開発するための一定の手順とはどんなものなのでしょうか．そのなかで設計作業はどのような意味，重要性をもっているのでしょうか．まず，このあたりから整理していくことにします．

プログラムの開発過程は，一連の変換過程とみることができます．すなわち，利用者の要求をコンピュータで稼動するプログラムに変換していく過程です．

まず，利用者の要求を把握し，そのプログラムに対する要件として定義することが先決です．

〔1〕要件定義

すべてのプログラムは，それを利用する利用者がいるはずです．その利用者が，プログラムに何を望んでいるかを的確に把握することが，プログラム作成の第一歩です．病院の患者監視プログラムの利用者は，入院患者の異常をリアルタイムでキャッチし，それを看護婦に知らせる機能を望むでしょう．その場合，何をもって患者の異常とみなすのでしょうか．血圧，脈拍，体温などいろいろな要因のうち，どれを監視の対象にすればよいのでしょうか．これらの事を的確に把握し，プログラムに対する要件として定義していかなければなりません．

〔2〕外部設計

要件定義ができれば，その要件をプログラムでどのような方法で実現するかを考えていくことになります．これがいわゆる設計作業であり，設計作業は大きく2つに分けることができます．外部設計と内部設計です．

外部設計は，利用者の立場からみたプログラムの設計です．換言すれば，利用者とプログラムとの間のインターフェースの設計です．わかりやすく言えば，利用者のために，そのプログラムの使い方を設計することです．どんなデータを入力し，どんな出力がえられるか，入力から出力への変換過程で，どんなデータ・ファイルが必要になるのかなどを設計することになります．

先の患者監視プログラムの例では，入力として，一定時間ごとの患者の血圧値，体温，出力として，それらの異常値，測定したデータが異常であるかどうかを判定するための正常値ファイルの設計などを行うことになります．外部設計のやり方いかんで，プログラムの使い勝手のよしあしがきまります．

〔3〕内部設計

プログラムの使い方が設計できれば，つぎは，そのような使い方を可能にするために，プログラムの構造をどのように設計するかを考えます．すなわち，プログラムをどうモジュール化し，モジュール間のインターフェースをどう設計するかが内部設計の主要作業になります．

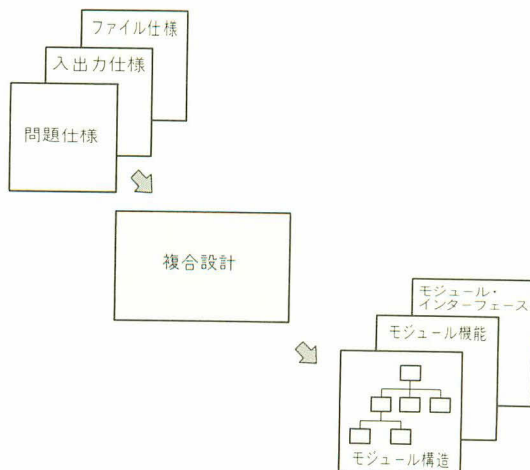
この過程が，プログラムの構造化設計であり，いいかえれば，プログラムの構造化設計とは，プログラムが何をなすべきかの仕様が定義されたとき，プログラムの総体的な構造を設計するための手法です．

構造化設計の入力となるのは，外部設計の結果であり，出力は，プログラムのモジュール構造と各モジュールの機能の定義，そしてモジュール間インターフェースの３つということになります(図1.17)．

このように，内部設計はプログラムの内部構造を決定する作業であり，そのやり方ひとつで，プログラムのわかりやすさ，結果として，プログラムの品質がきまります．

〔4〕プログラムの開発実施

内部設計が終了すれば，つぎはプログラムの開発実施です．ここでいうプログラムの開発実施とは，構造化設計によって定義された各モジュールの論理の設計とコーディング，

図1.17 構造化設計の入力と出力

そしてテストです，

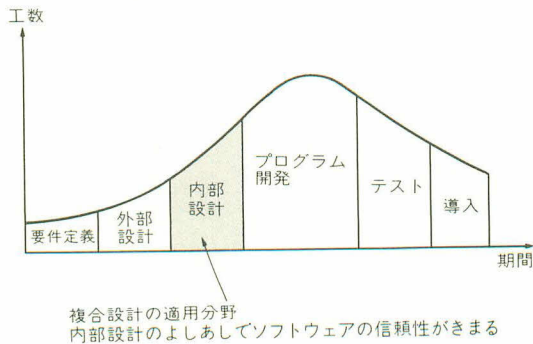
これらの作業で重要なのは，構造化定理の活用です．構造化定理とは，世にいう構造化プログラミングの基礎理論をなすものであり，プログラム論理を順次，選択，繰返しの3つの基本制御構造の組合せで設計しようとするものです(コラムA)．このことについては，第7章で詳しく検討することにします．

〔5〕テスト

プログラム開発時点で，個々にテストした各モジュールを，ひとつのプログラムとして統合し，プログラム全体として正しく稼動するかどうかをテストします．

テスト手順としては，トップ・ダウン・テスト，ボトム・アップ・テストなどがあります．このことについては，第8章で詳しく検討します．

プログラム開発過程の要約を図1.18に示します．

図1.18 ソフトウェアの作成過程

2．わかりやすいプログラム

プログラム設計のねらいは，わかりやすいプログラムを作成することにあります．わかりやすいプログラムはエラーを減少させます．すなわち，信頼性の高い高品質のプログラムを実現させます．

▶複雑さを減少させる３つの手段

それでは，どうすればわかりやすいプログラムになるのでしょうか．再び簡単な例で考えてみましょう．

図2.1　好ましくないプログラムの例

```
main()
  long ppm[60], pol;
  int   time, fcode, ecode;
  int   *ppol, *pecode;
{
  ppol=&pol;
  pecode=&ecode;
      :
  fcode=0;
  table (fcode, time, *ppol, *pecode, ppm);
      :
  fcode=1;
  table (fcode, time, *ppol, *pecode, ppm);
      :
  fcode=2;
  table (fcode, time, *ppol, *pecode, ppm);
      :
  fcode=3;
  table (fcode, time, *ppol, *pecode, ppm);
    :
}
```

図2.1に好ましくないプログラムの例を示します．この例では，好ましくない部分だけを強調するために，他の部分は省略してあります．このプログラムでは，テーブルppmに対する操作のすべてを関数tableによって行います．

関数tableは5つのインターフェース・データ(引数)をもっています．最初の引数fcodeは機能コードであり，その値が0のときはppmのクリア，1のときはppmへのデータ追加，2のときはデータの削除，3のときはデータの参照を指示しています．

2番目と3番目の引数は，ppmに対する2種類のデータであり，timeは時間，polはその時間で測定した大気の汚染度であるとします．

4番目のインターフェース・データはecodeです．この4番目のインターフェース・データecodeは，エラー・コードであり，関数tableでの処理に対して，エラーが発生したとき，エラーの種類によって値が設定されます．5番目のインターフェース・データは操作対象になっているテーブルppmです．

さて，このプログラムで何が好ましくないのでしょうか．一見するかぎり，テーブルの操作を一つの関数でまとめて行うように設計されており，設計者の苦労のあとが推察できます．しかし，関数tableの各引数について注意を向けると，混乱が生じます．

fcode=0のときのtime，pol，ecodeはどんな意味があるのでしょうが，ppmをクリアするときは，じつは，これらの引数は，とくに意味をもちません．つまり，ダミーの引数なのです．

fcode=1のときはどうでしょうか．このときは，ppmへデータを入力します．したがって，time，polは入力引数になります．入力引数として指定したtime，polの値がテーブルppmへ追加されます．ecodeはppmのテーブル・オーバフローが発生したときに設定されるでしょう．

fcode=2は，ppmにあるデータのうち，指定されたものを削除します．2番目の引数timeで削除すべきデータを指定します．3番目の引数polは，この場合特に意味をもちません．すなわち，ダミー引数になります．ecodeは，timeで指定された削除データがテーブルppmに見つからなかったとき設定されます．

fcode=3は，ppmの値を参照します．このとき，引数timeがキー入力になり，その時間の汚染度の値が出力としてpolに戻されます．ecodeは該当データが見つからなかったときに設定されることになります．

このようにみてくると，関数tableを使用するときには，細心の注意が必要になることがわかります．使用目的によって，同じ引数がダミーになったり，入力になったり，出力

になったりします．また，ecodeの値の意味もそのつど異なってきます．少なくとも，使用目的ごとに異なる引数の意味するところを，使用者は全部理解してからでないとうまく使いこなせないことになります．

このように使用時に細心の注意を必要とすることは，それだけエラーの発生の確率を高め，信頼性をそこねることになります．

しかし，このような例は，現実のプログラムで数多く使用されています．

この例のようなプログラム設計は，設計者の労力にもかかわらず，事態を複雑にしてしまいます．

わかりやすいプログラムを作るためには，この例がひきおこしたような複雑さを排除するよう考えていけばよいのです．それでは，どうすれば複雑さを排除できるのでしょうか．

プログラムに限らず，どんな種類のシステムでも共通に利用できる，複雑さを減少させるための3つの手段があるといわれています．それは，

- 分割すること
- 独立性を高めること
- 階層構造化すること

の3つです．

それでは，この3つの考え方について，もう少し詳しくみてみることにしましょう．

2.1 小さく分割することを考えよう

プログラムを複雑にする要素は何なのでしょう．第一の要素は，その大きさです．プログラムが大きくなればなるほど，そのプログラムを理解するために，人間が同時にたどり続けなければならない要素の数は増えます．要素の数が増せば，それだけ理解しにくくなります．ある調査では，120個のプログラムの大きさの平均は853ステートメント(命令)であり，1個のプログラムで使用した変数の数は384個であったそうです．そして，それらの変数が，そのプログラム内で平均1,195回出てきたとのことです．

このことからみても，1個のプログラムの内容を理解することが容易ではないことがわかります．数百にもおよぶ変数の意味を正しく理解しながら，論理をたどることになります．まして，論理が入りくんだ複雑なものであったらどうでしょう．

このような複雑さを減少させる手段は，プログラムを小さな単位，すなわち，モジュール(コラムB参照)に分割することです．小さくすれば，同時にたどらなければならない要

素の数が減り，それだけわかりやすさが増します．

プログラムをモジュール化すれば，必然的にモジュール間にインターフェースを発生させます．明確に定義されたインターフェースは，プログラムをわかりやすくするのに役立ちます．なぜなら，インターフェースはどのデータがどこに関係するか，しないかを示してくれるからです．それによって，注意の焦点をどこに絞ればよいかがわかります．

ここで問題になるのは，その分割の仕方です．分割の方法をまちがえば，むしろ，あらたな原因で複雑さが増加することもあります．たとえば，1つのモジュールに多くの異なった機能を含めたり，逆に，共通の機能を多くのモジュールに分散させたりしたばあいです．また，モジュール間のインターフェースが複雑で，複数のモジュールが期待されない方法で相互に影響しあっているようなときも複雑さは増加してしまうでしょう．

このようなことが生じないように，分割(モジュール化)時には考慮しておかなければならない点があります．それが第二の独立性を高めるという手段になります．

コラムB

モジュールとサブルーチン

プログラムの構造化設計を行うときの基本構成要素として，モジュールは大変重要な役割をになっています．似たような考え方として，サブルーチンという言葉があります．両者はどこが同じで，どこが違っているのでしょうか．

両者とも，プログラムの一部分を構成する要素であり，開始点と終了点がはっきりしていてプログラムの他の部分と区分けされています．そして，必要に応じて他の部分から呼出し可能であり，そのために固有の名前がつけられています．

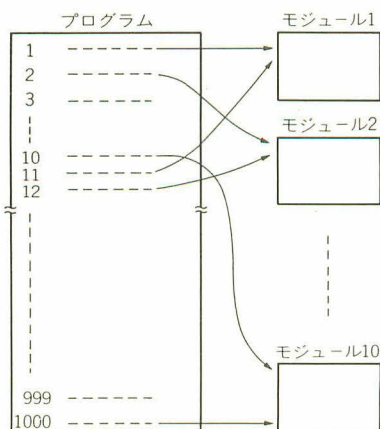
ただ，サブルーチンは場合は，そのプログラム内だけで使用されるのに対し，モジュールは他のプログラムでも使用可能です．モジュールは別個の独立した1個のプログラムであり，単独でコンパイルされるものです．それに対しサブルーチンはそのプログラムの他の部分と必ず一緒にコンパイルされます．

2.2 独立性を高くしよう

大きなものをいくつかの小さな単位に分け，個々の単位で考えていこうとするのが分割の概念でした．このとき，個々の分割した単位(モジュール)が，他のものから，できるだけ独立した存在になっていれば，他のことを考えずに自分の範囲内だけで物事の処理が可能になります．

このことを1つの例で考えてみましょう．いま，手もとに1000ステートメントからなるプログラムがあったとします．さきほどの調査にもあるとおり，おそらく，このプログラムは数百個の変数をもっていることでしょう．そして，このプログラムを理解するためには，大変な労力を必要とするでしょう．そこで，分割の手段を用いていくつかのモジュールに分割することにします．理解を容易にするために，100ステートメントからなる10個のモジュールに分割することにします．

分割の方法としてはいろいろ考えられます．1つの方法として，プログラムの1000個のステートメントの頭から順番に番号をふり，最初のステートメントを1番目のモジュールに，2番目のステートメントを2番目のモジュールに…といった具合に割り振り，11番目のステートメントになると，ふたたび1番目のモジュールに割り振っていくと，結果として100ステートメントからなる10個のモジュールに分割できます(図2.2)

図2.2 よくないモジュール分割

このような分割を行ったばあい，つぎのような結果が生じるはずです．

●モジュール間の関連性が強くなり，ひじょうにはんざつになる．各モジュール内のステートメントは，自身のモジュール内のステートメントより，他のモジュール内のステートメントとと強い関連性をもつようになる．

●もとのプログラムで行っていたいくつかの機能は，10個のモジュールに分散してしまう．見方をかえれば，各モジュールはいくつかの機能の一部分だけを実行することになる．

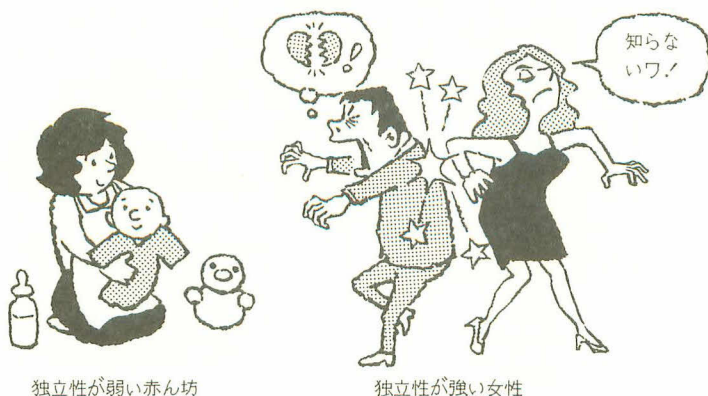
作成したモジュールが，このような特性をもったばあい，つぎのような問題が発生します．

●プログラムで何をやろうとしているのか理解しにくい．プログラムで実行するある機能を調べるためには，おそらく，分割した10個のモジュール全部を調べなければならない．

●プログラムの保守がむつかしくなる．プログラムのある機能に対して変更が必要になったとき，その変更は，すべてのモジュールに影響することになる．

●新しいプログラムを作るとき，これらのモジュールの1つを再使用することができない．なぜなら，それぞれのモジュールで実行している機能を特定できないからである．モジュールの再利用をはかる基本的な動機は，ある機能を生かすことである．機能を特定できないモジュールの再利用を考える人はいない．

図2.3　独立性の概念

独立性が弱い赤ん坊　　独立性が強い女性

▶モジュールの結合度と強度

これらの事実から，このような分割の方法がまちがっていることは明白です．そして，結果から，逆に分割はつぎのような方法で行うべきであることがわかります．

●各モジュール間の関連性を最小にする．

●個々のモジュール内のステートメントの関連性を最大にする．

この2つの事実こそ，独立性の概念です(図2.3)．分割に際して，高い独立性を達成させるためには，関連性の低いステートメントは，別々のモジュールに分けるべきです．また，関連性の高いステートメントは同じモジュールに入れるようにすればよいのです．

構造化設計では，前者の考え方をモジュール間結合度，前者をモジュール強度と呼び，結合度を最小にし，強度を最大にする方法を明示しています．強度，結合度については，この後第4章および第5章で詳しく解説することにします．

2.3 階層構造化をはかろう

複雑さを減少させる第三の手段は，階層構造化の概念です．階層構造化の概念は，プログラムを構築したり，理解したりするとき非常に重要です．

先にも述べたように，プログラムを複雑にしている原因の1つは，同時に考えなければならない要因の数の多さにあります．ひとりの人間の頭で扱える要因の数は，ごく限られたものであり，その範囲内で問題の解決をはかるためには，物事を小さい単位に分割して，個々の単位で独立して考えることを可能にすることと，もう1つは物事を階層構造的に定義していくことです．階層的なものの見方は，人間の注意の焦点をどこに合わせればよいかを指図することになり，理解を助けるのです．階層構造の各レベルは，より下位レベル要素間の関連性を集約したものです．このことは，そのレベルでは不必要に詳細な情報をかくして(情報隠匿の概念)，要約的にものごとを考えることを可能にすることを意味しています．人間は作成しなければならないプログラムの大きさが，大きくなればなるほど，それを一挙に作ることに困難さを感じます．考えなければならない多くの要因が同時に提示されるからです．

その意味で，階層化して物事を考えるのは，概要から詳細へと段階的に整理することを可能にし，有効です(図2.4)．

プログラムの設計にさいして，モジュール化をはかり，個々のモジュールの独立性を高めたとしても，それらは，最終的には一つのプログラムへとまとめられなければなりませ

図2.4　階層構造の概念(それぞれのレベルでの役割がある)

会社方針を
きめよう
社長
部の目標を
きめよう
部長
部長
部長
課で何を
するべきか
課長
課長
課長
私の
やる仕事
は-----
担当者
担当者
担当者

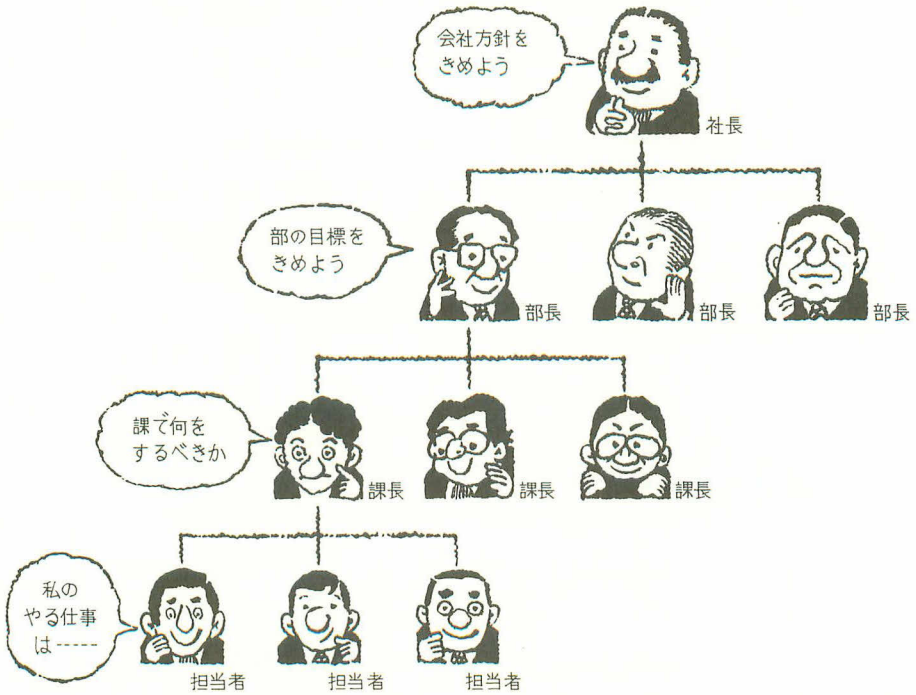
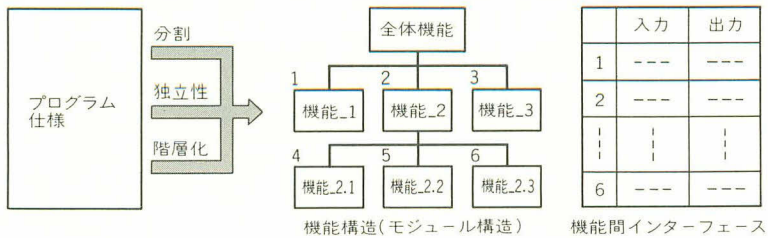



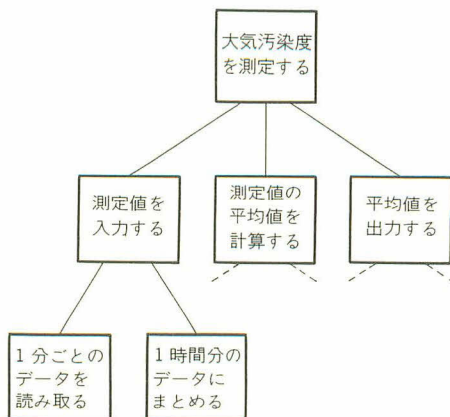
図2.5　よい設計のための3つの手段

ん．そのことをふまえて，最初にモジュール化をはかるときに，個々のモジュールを階層的に分割するようにすればよいのです(図2.5)．それによって，モジュール化がやりやすくなり，かつ，プログラムの理解度が向上することになります．

プログラムをモジュールの階層構造に分けるということは，換言すれば，プログラムの機能をトップダウンで展開し，結果としてえられる機能階層構造をモジュール構造に対応させることなのです．

このことは，独立性の概念とも密接な関係をもっています．詳しくは後の章であらためて考えることにします．

ここでは，階層構造化ということを具体的イメージとして理解していただくために，簡単な例を紹介するにとどめておきましょう．

図2.6は，大気汚染度の測定問題に対する最上位レベルの階層構造です．この問題の全体機能は，「大気汚染度を測定する」です．この全体機能は，次のレベルで，「測定値を入力する」，「測定値の平均値を計算する」，「平均値を出力する」の3つの機能に分割されています．これは，大気汚染度を測定するためには，この3つの機能を行なう必要があることを示しています．また，逆の言い方をすれば，この3つの機能を集約したものが，全体機能として，大気汚染度を測定するになるということです．ここまで明らかになれば，必要に応じて，さらに，次のレベルへと詳細化していけばよいのです．たとえば，「測定

図2.6 階層構造の例

値を入力する」機能は，「1分ごとにデータを読取る」，「1時間分のデータにまとめる」といった具合に….

このことから，概要から詳細へ，レベルごとに注意を集中することで，問題の理解を容易にし，わかりやすさを高めていくことが理解していただけるはずです.

プログラムの設計にさいして，これら3つの概念，分割，独立性，階層構造化を具体的にどのように実現していくかは，後の各章で順を追って説明します.

3．プログラム設計のための用語を定義し，表記法を身につけよう

わかりやすいプログラムを設計する第一歩は，分割することでした．要件と外部仕様をもとに，プログラムを一枚岩のように設計するのではなく，いくつかのモジュールに分割して設計し，それらのモジュールが最終的には，1つのプログラムとして，ルールにそった稼動ができるようにすることです．すなわち，分解と合成の過程が必要になります．

その意味で，分割は分解の過程であり，階層構造化は合成の過程ということもできます．

このことをソフトウェアの設計過程で可能にするためには，モジュールというものを正確に定義し，その使い方に対するルールをしっかりと設けておく必要があります．

3.1 モジュールを正しく定義し，その用法を理解しよう

モジュールは，プログラムの基本単位です．実行可能なプログラム命令の集合であり，つぎのような規準に合致するものであるとします．

(1) 実行可能な複数のプログラム命令が語彙としてまとまっている．これらの命令群は，物理的に一緒になっていて，境界識別子(はじめと終わりを明確に示し，他との区別を可能にさせるもの)で区切られている．

(2) これらの命令群は，固有の名前をもつことができ，他からその名前で呼び出すことができる．

(3) 単独にコンパイルできる(図3.1(a))

このことから，モジュールは，プログラムを構造的にながめたときの基本単位とみることができます．たとえば，Fortran言語では，サブルーチン/サブプログラムがモジュールに該当します．図3.1(b)はサブルーチン/サブプログラムの形式を示しています．

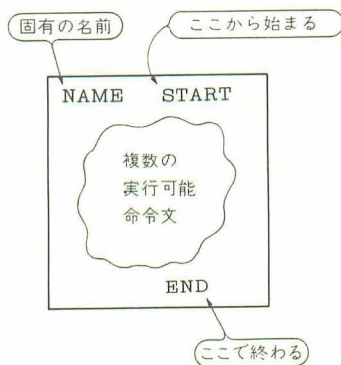
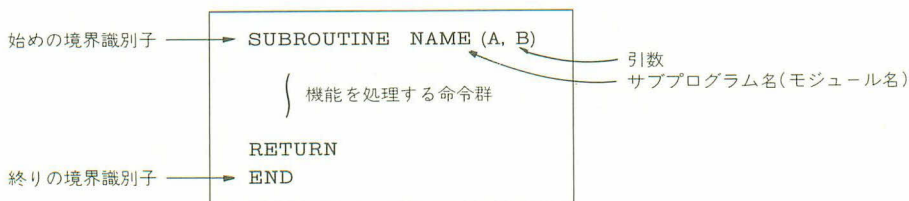
図3.1 モジュールの構造

（a）モジュールの概念

（b）モジュールの例

図3.1(b)で，最初に出てくるSUBROUTINEステートメントがモジュールとしての始めの境界識別子になります．また，終りの境界識別子はENDステートメントであり，この間にある命令群がモジュールの中味になります．

また，SUBROUTINEステートメントに出てくるNAMEはモジュール名となり，このモジュールを他から呼出すときに使われます．

また，サブルーチン・サブプログラムは，他のモジュールとやりとりするデータを引数の形で指定することができます．引数としては，このモジュールに入力されるデータ，あるいはこのモジュールから出力されるデータを必要な数だけ指定できます．

C言語では，モジュールに該当するのは関数です．

C言語の関数は一般的に次のような形式をもっています．

```
関数名（引数リスト）
引数宣言
{


}
```

ここで，関数名はその関数固有の名前であり，プログラムでその関数が必要になったときは，その名前により呼び出すことができます．これは先の(2)の条件を満たします．また，

関数名(引数リスト)

はその関数がそこから始まることも示しています．また，}はその関数の終りを明確に指示します．そして，{}の間にその関数が行うべきことを実行するための命令群が語彙としてまとまっています．それらの命令群は{}の間にすべて存在するわけですから物理的にも一緒の場所にあるとみなすことができます．これらのことから(1)の条件も満足することもわかるはずです．

ただ，気をつけていただきたいのは，関数はその性格上，原則として出力は1つしか返せないことです．モジュールを一つの固有機能を実行する単位ととらえれば，その機能を実行した結果として，複数種類の出力を生み出すことは当然考えられます．

FORTRANのサブルーチン・サブプログラムでは，引数として複数の出力を指定できますので，ごく自然に処理できますが，Cの関数ではこのあたりに工夫が必要になります．

たとえば，出力データのアドレス(ポインタ)を引数に指定するといった方法が考えられます．

```
funcn(in1, in2, pout1, pout2)
  int in1, in2;
  int *pout1, *pout2;
   {
     :
     *pout1=
     :
     *pout2=
     :
   }
```

この例では，pout1，pout2が出力を戻す引数であり，この関数を呼出す関数(たとえば，main関数)でfuncn(in1, in2, &out1, &out2)と指定すれば，out1，out2に出力が戻されます．なお，関数は特定のプログラム内だけで使用するのなら，そのプログラムと一緒にコンパイルすることになるでしょうが，他のプログラムでも使用する場合は，別のソース・ファイルとして単独にコンパイルすることも可能です．その意味で，関数はモジュールの定義の(3)の条件をも満たしています．

関数は引数リストを指定できることにも注目してください．わかりやすいプログラムを設計するための第1の条件，分割することの意義は，プログラムをなるべく小さい単位に分けて考えていけるようにすることと同時に，分けた単位(モジュール)間でのインターフェースを明確にすることでした．インターフェースを明確にすることで，どのデータがどこで使われたり，使われなかったりするのかがよくわかり，それが問題全体をわかりやすくすると説明しましたが，引数を指定するのは，まさに，このインターフェースを明確にすることなのです(図3.2)．

つぎに，モジュールを使用するときのいくつかの約束をきめておきましょう．図3.3をよくみてください．この図はモジュールの使い方に対して基本的な事柄を示しています．

図3.2　インターフェースを明確に

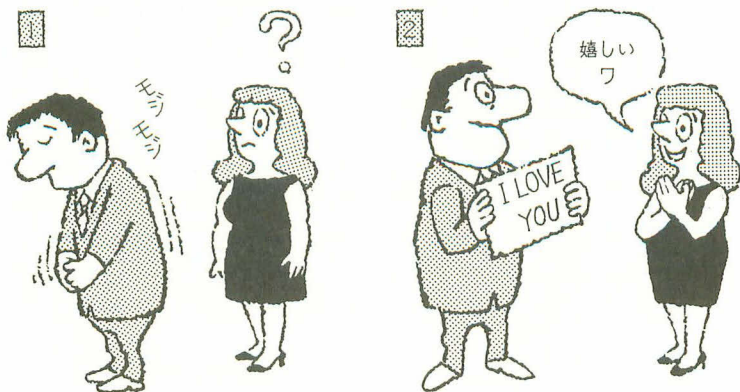



図3.3 モジュールの使い方に関する基本的事項

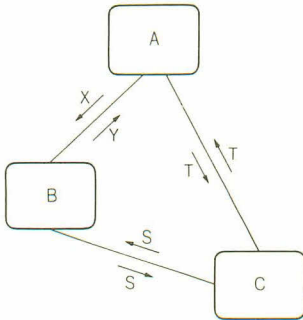
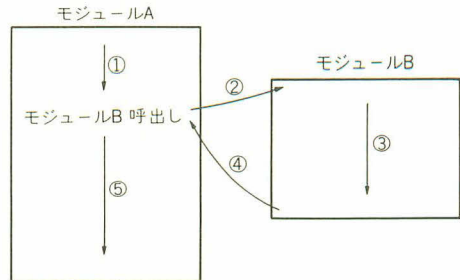



図3.4 モジュール実行時のルール

- ここには，3つのモジュール A，B，C がある．
- モジュール A は，モジュール B，C を呼び出す．
- モジュール B は，モジュール C を呼び出す．
- モジュール B は，X という入力を受けとり，出力として Y をかえす．
- モジュール C は，S または T という入力を受けとり，出力としてそれぞれに S または Y をかえす．
- モジュール B は，モジュール A に従属している．
- モジュール C は，モジュール A と B の両方に従属している．

これらの基本的事項から，モジュールが他のモジュールから固有の名前で呼び出すことができること，呼び出すときにモジュール間で入出力データをパラメータとして引渡し可能なこと，モジュール間には従属関係(階層化)が存在することがおわかりいただけるはずです．

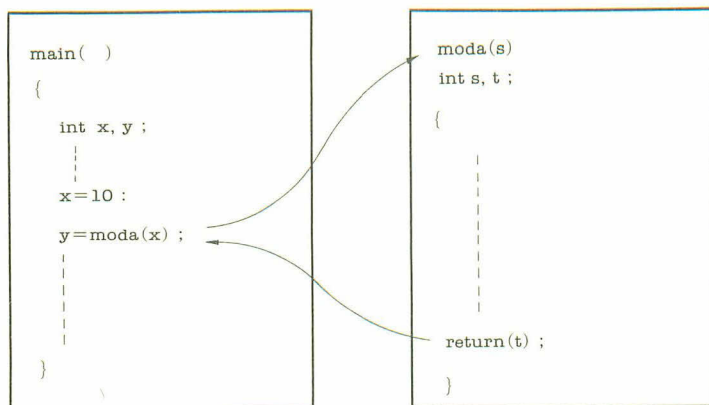
また，これらの事実のほかに，モジュールの用法として，つぎの規準もきめておきます．

あるモジュールが他のモジュールを呼び出したとき，呼び出したモジュールの実行は，呼び出されたモジューの実行が終わるまで待たされる．

呼び出されたモジュールの実行が終わると，実行は呼び出したモジュールに戻る．そして，呼出し命令のつぎの命令から実行する(図3.4)．

以上のことから，モジュールの正確な概念と，それらが集まって，1本のプログラムとして稼動する仕組みがわかっていただけたと思います．

ふたたび，C 言語の関数をとりあげてみますと，関数の用法はいま述べたモジュールに

図3.5　モジュール間の制御の流れ

関する用法をすべて満たしています．たとえば，図3.5 ようなプログラムがあったとします

このプログラムは，関数間の制御の流れを説明するのが目的ですので，目的外の詳細部分は省略してあります．

さて，このプログラム例では，main 関数と moda 関数が存在します．このプログラムが実行するときは，まず main 関数の最初の部分から始まり，x=10 の命令を実行した後，制御は moda に移ります．moda では所定の論理を実行し，最後の return 命令を実行後，制御を main に戻します．図3.5 はこのことを図示しており，これは図3.3 のルールにそった実行の形であることが理解できます．

3.2　機能と論理のちがいを正しく理解しておこう

いままでの説明で，モジュールとはどんなものなのかは正しく理解できたと思います．プログラムの構造化設計とは，1つのプログラムをいくつかのモジュールに分割し，それを階層構造にまとめあげる過程です．

その詳細に関しては，順を追って説明していきますが，その作業をはじめる前にぜひ知っておいてほしいのは，機能と論理の違いについてです．この違いを知ることは，プログラムの構造化設計を進めていくうえで大変重要なことなのです．

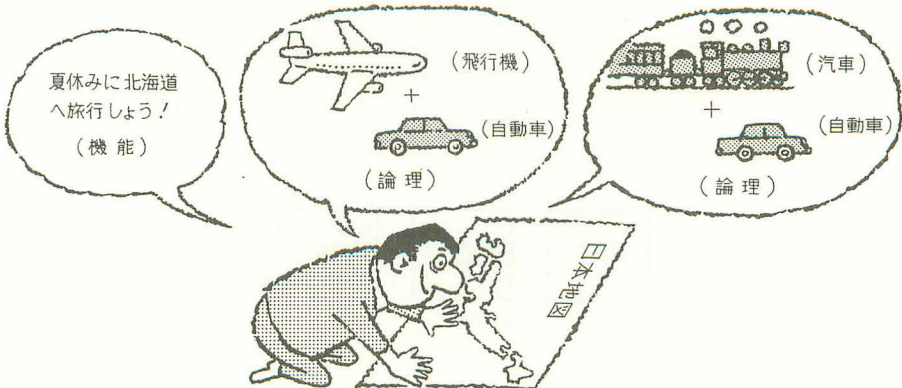
構造化設計では，まずプログラムをモジュールに分割するときに，個々のモジュールの

機能を定義します．そのあと，モジュールの論理を設計します．

ここでいうモジュールの機能とは，そのモジュールがなすべきことです．一方，モジュールの論理とは，モジュールがその機能をいかに実行するか，その手順を指します．

機能と論理の違いをより正しく理解するために，いくつかの例で考えてみましょう．たとえば，旅行をすることを例にとってみます．話をより具体的にするために，夏休みに北海道へ10日間の旅行をするものとします．この場合，機能は「10日間の北海道旅行」です．この北海道旅行を行う方法はいろいろあります．札幌まで飛行機で行き，その後，北海道内はレンタ・カーをかりて自動車でまわる．帰りは，また飛行機を利用する，あるいはすべての旅程に汽車を利用する，すべての旅程に自家用車を用いる，飛行機と汽車を利用する等々の方法が考えられます．ここであげたいくつかの例が北海道旅行をする機能の論理ということになります(図3.6)．

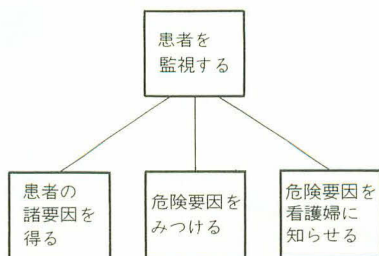
このように，ある特定の機能を行うためには，いくつかの論理が存在します．もう1つ別の例をとりあげてみましょう．時計について考えてみます．時計の機能は，その時その時の正確な時間を示すことです．そして，この機能を行うためには，いろいろな方法があります．クオーツで知られているように，水晶発振器を用いる方法もあれば，いまはほとんどみられなくなりましたが，スプリングを用いたネジ巻き時計もありましたね．また，昔は，日時計，砂時計などもありました．これらは，時間を示すためのいくつかの方法であり，論理なのです．

図3.6 機能と論理

プログラムの世界に話を戻しましょう．プログラムで行うべきことは，通常，プログラムの機能仕様書としてまとめます．そのプログラムで何を行おうとしているのか，それがプログラムの機能なのです．そして，その機能を行うための手順を，C言語やBASICといったプログラミング言語でコーディングしていく，これがプログラムの論理の記述なのです．同じ機能を行うにも，作る人が異なれば，いろいろな論理のプログラムができるのは，読者のみなさんのほとんどは経験済のことでしょう．

機能と論理の基本的なちがいは理解していただけたと思います．しかし，もう少し，機能と論理について続けて考えてみましょう．

前章で，わかりやすいプログラムに設計するには，プログラムを分割すること，独立性を高くすること，階層構造化することの3つを守ることであることを説明しました．そして，これらの3つの概念は，お互いに強い関連性をもっています．すなわち，独立性を高くして分割するためには，プログラムを機能的にみて分割する必要があり，機能的に分割していけば，結果として，機能が階層化され，それがプログラムのモジュール構造になっていくといったことです．

このことから，これら3つの概念の関連性の基盤となっているのが，機能であることがわかります．そして，機能はより小さないくつかの機能に分割でき，それらが階層化されるということがわかります．1つの例でみてみましょう．図3.7は，病院の入院患者を監視するプログラムを機能階層化した例です．このプログラムの全体機能は，患者を監視するです．そして，この機能を行うために，次の3つの機能を行う必要があります．

図3.7 「患者を監視する」の機能分割

●「患者の諸要因を得る」

●「危険要因をみつける」

●「危険要因を看護婦に知らせる」

「患者を監視する」機能は，入院患者の体温とか血圧，脈拍などの要因を自動測定し，それらの要因の値に異常があれば，すぐにそれを看護婦に知らせることをさしています．そして，これらの三つの機能，たとえば，「危険要因を見つける」機能は，その患者の諸要因の正常値と測定値とを比較し，測定値が正常範囲外であれば，それが危険要因として認められることになります(図3.8)．

この問題に対する詳細分析は，後の章で再度あらためて取りあげますが，図3.7～図3.8の例から，機能を階層化するという意味が十分理解できるはずです．また，機能を階層的に分割することが，結果として，独立性を高くしていることにもなっています．このことは，第4章で詳しくみてみることにします．

ここでは，階層化したときの機能においては，あるレベルの機能は，その機能に従属する(下位レベル)すべての機能を包含したものであることを十分くみとっておいて下さい．そして，このことは，機能をモジュールという用語におきかえても，そのままあてはまることなのです．

すなわち，モジュールの機能は，そのモジュールに従属するすべてのモジュールの機能を包含します．

ただ，ここで気を付けていただきたいのは，

モジュール機能＝Σ従属モジュールの機能

ではないということです．これは，上のモジュール機能についての文章的な説明と矛盾するようですが，決してそうではないのです．これは図にすると明白になります．図3.9は，

図3.8 「危険要因をみつける」の機能分割

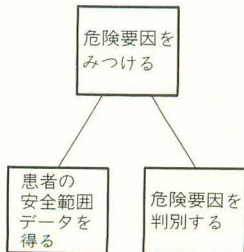
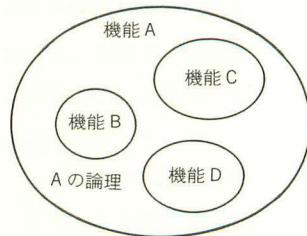



図3.9 A＝B＋C＋D＋(A 自身の論理)

そのことを説明しています．この図で，機能Aは，従属する3つの機能B,C,Dを包含しますが，B,C,Dの3つの機能でAのすべてを表わしているわけではありません．機能Aは，機能B,C,Dで表わせる部分と，その他に自分自身の独自の論理部分をもっています．

したがって，次のように言うのが，モジュール機能に対する正しい定義になります．

「あるモジュールの機能とは，そのモジュール自身の論理とすべての直接従属するモジュー機能とをまとめた一つの機能である」

このことをC言語プログラムを例にとってみればつぎのようになります．

```
main()
{
        ⋮  Ⓐ
    b=funcb(b1, b2);
        ⋮
    c=funcc(c1, c2);
        ⋮
    d=funcd(d1, d2);
        ⋮
}
```

この例で，main関数(モジュール)はA機能を表わしています．このmain関数は，自分自身の論理(点線部分Ⓐ)と従属する3つの関数(モジュール)funcb，funcc，funcdから構成されています．この3つの従属関数がそれぞれのB，C，Dを表わしています．

これまでの説明で，いまや機能と論理という言葉がプログラム設計の分野で，どんな概念を表わしているのか十分ご理解いただけたはずです．

ここで，この節の冒頭で述べたことをもう一度確認して下さい．プログラムの構造化設計とは，まず最初に，プログラムをモジュール分割することです．そして，モジュールの機能を定義します．つぎに，個々のモジュールの論理を設計していきます．

モジュールの機能を定義するときは，それが論理的な表現にならないように気をつける必要があります．たとえば，主制御モジュールといった表現は使わないほうがよいのです．主制御という言語からうける印象としては，機能的なものより論理的なものを頭に描いてしまいがちだからです．

機能的な表現をする簡単な方法は，機能の定義を一つの目的語と一つの動詞を用いて行ってみることです．「患者を監視する」，「危険要因を見つける」などの表現は，この形

で表わされていることにあらためて気づかれることでしょう．

3.3　標準的な表記法をきめておこう

プログラムの構造化設計を実際に進めていくときに，思考を明確にし，その結果を誰にでもわかりやすいものとして表現するための技法があれば大変役にたちます．そのための技法として，構造化チャートが一般的に使われます．

構造化チャートは，機能あるいはモジュールを階層的に表現するための図的技法であり，プログラムを階層構造化した1組のモジュールとして表現すると同時にモジュール間のインターフェースの定義も行えるようになっています．手順(論理)は，原則として，表現せず，プログラムの静的な階層構造を表現するのがねらいです．

構造化チャートの基本構成要素は，長方形の枠とその長方形を結びつける連係線です．長方形枠は一つのモジュールを表わします．枠の内部にモジュールの名前(機能)を書きます．

大気汚染度
を測定する

前もって作られた既存のモジュールを使用するときは，それが既存モジュールであることを示すために，つぎのような縦線が2本の長方形枠を用います．

平方根を計算する

プログラムの内容によっては，複数箇所で同じ機能(モジュール)を実行することがあります．たとえば，複数箇所でエラー・チェックを行い，エラーがあれば，エラー・メッセージを書出す場合などです．エラーの内容は箇所によってちがいがあっても，エラー・メッセージを書き出す仕事は，おそらく共通化できるでしょう．そのような場合は，共通モジュールとして，1つ作り，必要な箇所でそのモジュールを呼出し，使用します．共通モジュールであることは，長方形枠の右肩をぬりつぶすことによって表わします．

エラー・メッセ
ージを書き出す

図3.10 共通モジュールの使用

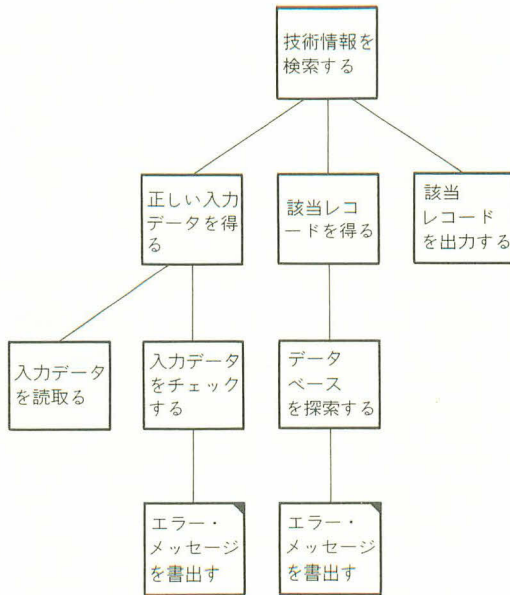
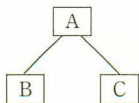
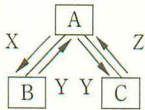



図3.10は，共通モジュールがモジュール構造図で使われている例です．正しい入力データが得られなかったときと該当レコードがデータベースに見つからなかったとき，エラー・メッセージが書出されます．エラー・メッセージの内容はちがってもメッセージを書出す方法は同じですので共通モジュールが使用されています．

モジュール間に存在する連係線は，モジュールの呼出しを意味します．長方形の底辺から出ている線は，そのモジュールが他のモジュールを呼出すことを表わしています．逆に，長方形の上辺への連係線は，他のモジュールがそのモジュールを呼び出していることを表わしています．

上の図で，モジュールAは，モジュールBとCを呼び出します．換言すれば，モジュールB，Cは，モジュールAに呼び出されます．

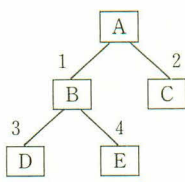
ただ，この図では，AがBとCを何回呼び出すとか，どんなときに呼び出すとかまでは表現していないことに気を付けて下さい．それらは論理の問題であり，構造化チャートで表現するのは，原則として，機能の階層関係だけなのです．

プログラムの構造化設計の表現でもう1つの重要なことは，モジュール間インターフェースの表現です．

上の図で，連係線上の矢印と記号がインターフェースを表現しています．XはAからBへわたされるデータであり，YはBからAへわたされるデータを表わしています．また，YはAからCへ，ZがCからAにわたされます．この関係をC言語プログラムで表現すれば，つぎのようになるでしょう．

```
main( )
{
  int x, y, z;
     ⋮
  y=funcb(x);
     ⋮
  z=funcc(y);
     ⋮
}
```

```
funcb(s)
  int s, t;
{
     ⋮
  return(t);
}
```

```
funcc(u)
  int u, v;
{
  int v;
     ⋮
  return(v);
}
```

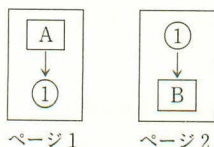
この例では，インターフェースとしての入出力パラメータの数が少なく，したがって，表現も簡単でみやすい形になっています．しかし，実際のプログラムでは，モジュール構造はもっと複雑になり，また，インターフェース・データの数も増加します．そのようなときに，上記の表現法では，図が大変繁雑になり，みにくくなってしまいますので，インターフェース表現に関しては，つぎのような方法にするのがよいでしょう．

	入力	出力
1	X	Y
2	Y	Z
3	S	T
4	U	T

この方式では，モジュール構造図上では，インターフェースは，直接データを記入せず，番号だけを記述します．そして，別のところに，インターフェース表として，それぞれの番号のインターフェースと入出力データを記述します．この表において，入力，出力は呼び出されるモジュールの観点から定義されていることに注意して下さい(図3.11)．たとえば，インターフェース1の入力Xは，AからBへ入力されるデータであり，出力Yは，BからAへ出力されるデータを意味します．

以上が，構造化チャートを用いて，プログラムの構造化設計を表現する場合の表記法ですが，これらの約束ごとに加えて，つぎのような規則を守るようにすれば，わかりやすさがさらに増すことになります．

図3.11　子供のインターフェース

●プログラムの全体図は，なるべく１枚の用紙上に書く．

どうしても，１枚に書ききれないときは，別のページとの連係を明確にしておく．たとえば，下のような接続子を用いる．

●同上用紙上での連係線の交差が多くなるときは，上のような接続子を用いてもよい．ただ，接続子の使用が多くなると，かえって，みにくくなることがあるので注意すること．

●用紙上のモジュールの位置は，原則としては，プログラム構造上のそのモジュールの位置とは直接関係ない．しかし，できれば，用紙上で，上位モジュールの下にその従属モジュールがくるように書けば，その階層関係が理解しやすくなる．

あるモジュールが複数個の従属モジュールを有する場合，従属モジュール間の横(階層構造上での同じレベル)の関係は，原則としては，何も意味しない．しかし，親モジュールがそれらの従属モジュールを呼び出す順序について，その段階で予測できるなら，その順序(左から右)にならべておくのがよい．

3.4　表記法を用いていくつかの例を描いてみよう

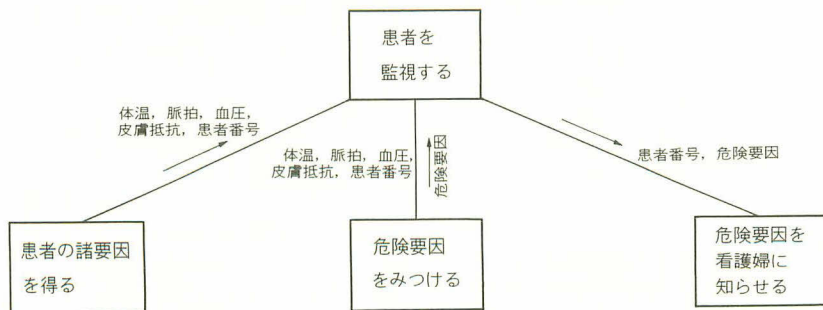
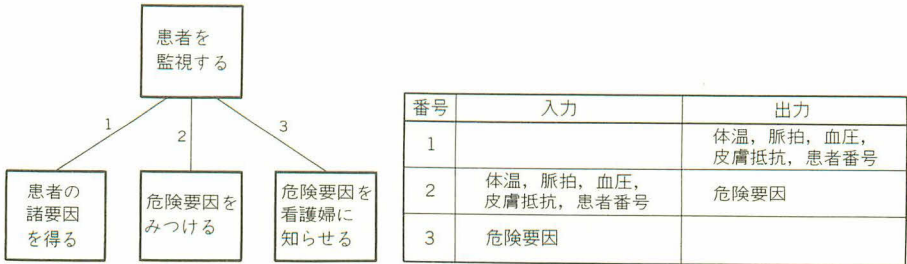
それでは，構造化チャートを用いていくつか簡単な問題を描き，その良否について検討してみましょう．

図3.12は，図3.7でとりあげた「患者を監視する」の機能構造図に機能間のインターフェース・データを書き加えたものです．これは構造化チャートの標準的な表記法にそって書いたものです．長方形枠はモジュール(機能)を表わしています．枠のなかに機能名を記述しています．モジュール間の連係線でモジュールの従属関係を表わしています．すなわち，「患者を監視する」モジュールは，「患者の諸要因を得る」，「危険要因をみつける」，「危険要因を看護婦に知らせる」の3つのモジュールを呼び出します．この図では，基本的には，この3つのモジュールを呼び出す順序とか回数，条件といったものは表現していませんが，前節で述べたように，常識的には，左から右への順で呼び出すと考えていいでしょう．機能的に考えても入力→処理→出力の順で左から右へならべることがわかりやすさにつながるはずです．

ただ，この描き方で問題なのは，インターフェース・データです．データの数が多くなると，どうしても繁雑になり，見にくくなりますね．やはり，このようなときは，図3.13のように，インターフェース表のなかにインターフェース・データをまとめる方がみやすくなります．また，図3.14のように，階層構造を横展開で描く方法もあります．たて長の用紙上に描くときは，同一レベルに多くのモジュールが存在するばあい，横展開の方がたて展開より書きやすい利点があります．しかし，モジュールや機能の従属関係を視覚的

図3.12 「患者を監視する」の構造化チャート

図3.13 図3.12の修正

番号	入力	出力
1		体温，脈拍，血圧，皮膚抵抗，患者番号
2	体温，脈拍，血圧，皮膚抵抗，患者番号	危険要因
3	危険要因	

にわかりやすくするのは，どちらかといえば，たて展開の方でしょうから，たて展開の方がよいか，よこ展開の方がよいのか，それは一長一短です．

本書では，原則として，たて展開方式を採用していきます．

図3.10には共通モジュールの描き方の例がでています．この例では，共通モジュールはそれを必要とする箇所で重複して描いています．用紙上のスペースに余裕があるときは，このように描いた方が理解しやすいでしょう．ただ，この描き方は同じモジュールを複数個書くことになりますから，その分用紙のスペースを費すことになり，限られた大きさの用紙に規模の大きな構造を描くときには向きません．

そのようなときは，共通モジュールは1個だけ書き，それを必要とするモジュールから

図3.14 図3.12を横展開したもの
(インターフェースは省略)

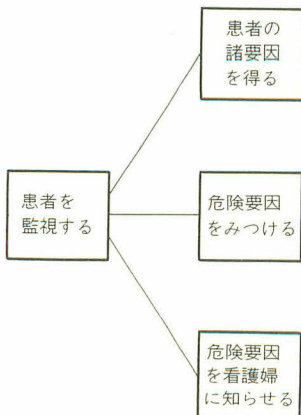



図3.15 共通モジュールのもう一つのかきかた
(インターフェースは省略)

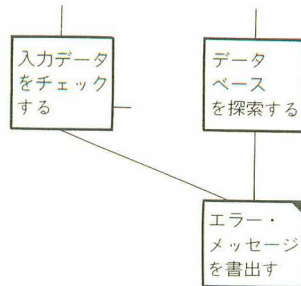
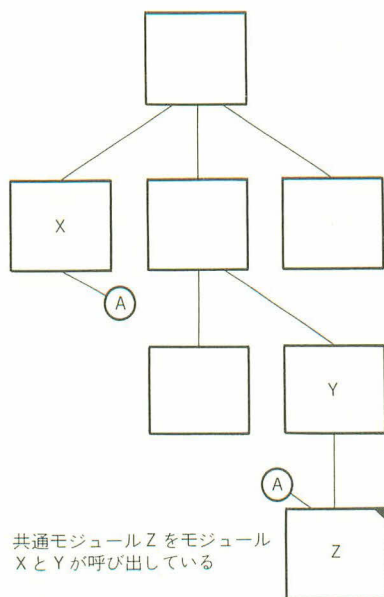



図3.16 離れた位置の共通モジュールを呼ぶときは連絡線を接続子で結ぶ

連係線で結ぶ描き方がよいでしょう．例を図3.15に示します．この例では，隣接する2つのモジュールから共通モジュールが呼び出されていて，連係線で結ぶときに大変都合がよくできています．実際には，このように都合がよいばあいばかりでなく，場所的に遠くはなれた複数個のモジュールが共通モジュールを呼び出すことがあります．そのばあいは，図3.16に示すように，連係線を接続子で結ぶか，連係線同志を交差させる方法をとります．

どの方式を採用するかの選択は，そのときの状況に応じて，どの方式が一番見やすいかを判断して決めることになります．もちろん，いくつかの方式を組合せてもかまいません．

4．モジュールの強度を強くすることを考えよう

プログラムの設計過程において，分割の基本単位はモジュールであり，モジュールに分割するときの重要な点は，独立性を強くすることでした．

モジュールの独立性を高めるためには，前述したように各モジュール内の関連性を最大にすることと，モジュール間の関連性を最小にすることの２つの面から考えていく必要があります．

モジュール強度は，前者に関連したものであり，あるモジュール内の要素間の関連性についての１つの尺度です．

プログラムのモジュール化は，ずいぶん昔から行われていたことであり，とくに，新しい概念というわけではありません．

ただ，モジュール化を行うときのはっきりした規準はなく，個人の考えにまかせていた傾向が強かったので，いろいろなやり方で行われてきたわけです．それらのやり方には，モジュール強度という観点からみて好ましいもの，好ましくないものが混在しています．

表4.1は，いままでのモジュール化のタイプをモジュール強度の面からみて分類，整理したものです．ここには，７つのタイプがあらわれていますが，強度の低い順に上から下へとならべてあります．暗合的強度は一番低く，機能的強度は一番高くなります．強度が高ければ独立性も高くなるわけですから，結論からすれば機能的強度が望ましいわけです．

ただ，何をもって強度が低いとか高いとかを判断するのかは，絶対的な基準が存在するわけではなく，各モジュールの相対的効果によって決めたものです．相対的効果とは，エラーの発生頻度，プログラムの保守のやりやすさ，モジュールの再使用性などの面から総合的に判断したものです．

モジュール強度というものをよりよく理解するために，いくつかの強度を例にとって考えてみましょう．

表4.1 モジュール強度一覧

	種類	定義	特徴
低い ↑	暗合的	・機能を定義できない． ・互いに関係のない複数個の機能を実行	・モジュール内の各要素間の関連性は低い．反対に，他のモジュール内の要素との関連性が高い． ・インターフェースが複雑． ・再使用，保守，拡張時に問題がある．
	論理的	・関連したいくつかの機能をふくみ，そのうちの一つが呼出しモジュールによって選択され，実行される． ・どの機能が選択されるかはインターフェースで指示される．	・単一のインターフェース，パラメータが機能ごとに解釈がちがったり，無視されたりする． ・一つの機能の使用に対しても，他機能の知識が必要． ・インターフェースの変更がすべての機能に影響する． ・機能コードとそのエラー・チェックが必要になる．
	時間的	・複数の逐次的機能を実行する．機能間の関連は時間的共通性以外にはほとんどない．	・モジュール内の複数機能間の関連性は低い．反対に，他モジュール機能との関連性が高い．
↓ 高い	手順的	・複数の逐次的機能を実行する．機能間には問題仕様にもとづく関連性を有する．	・とくに良い点もないが悪い点もない．
	連絡的	・手順的＋(機能間にデータの関連性を有する)．	・手順的よりは強度は強い．
	情報的	・複数入口点をもつ． ・各入口点は単一の固有の機能を行う． ・これらの機能はすべて共通のデータを扱う． ・いくつかの関連した機能的強度モジュールを一つにまとめたもの．	・データ構造の変更が一つのモジュールにしか影響しない． ・論理的強度の問題点が解消できる． ・独立性と拡張性が高く，エラー率が低い．
	機能的	・一つの固有の機能を実行する．	・強度は強い．

4.1 強度が弱くなってしまうモジュール化は，どんな時におきるのだろうか

〔1〕暗合的強度をもつモジュール

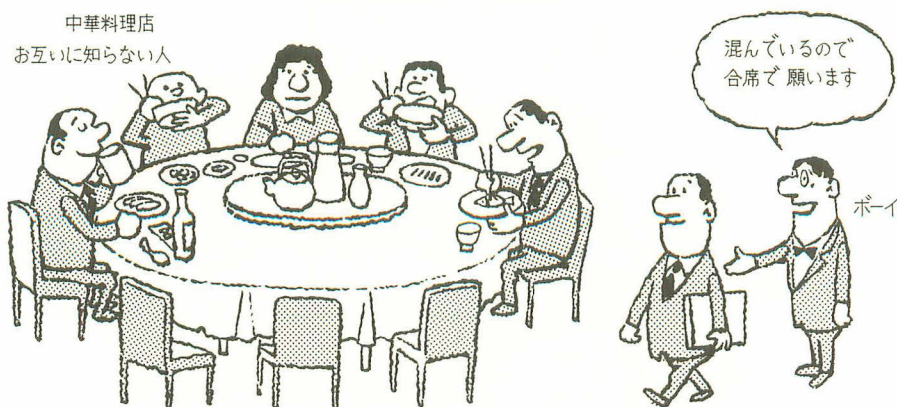
暗合的強度は，強度が一番低いタイプですから，モジュール化にさいしては一番やってはいけないケースです．しかし，よくないものを最初に説明すれば，逆によいものがどんなものかがよく理解できると思いますので，最初にとりあげてみます．

表4.1にその要約を示したように，「暗合的強度をもつモジュール」とは，つぎのような規準に合致するモジュールのことです(図4.1)．

●機能を定義することができない．

●複数のまったく関係のない機能を実行している．

すなわち，「このモジュールで何を実行しているのですか？」と聞かれても容易に説明

図4.1 暗号的強度は中華料理店の丸テーブル

できないようなモジュールです．先にとりあげた1000個のステートメントからなるプログラムを10個のモジュールに分割するときに，ステートメントに順次番号をつけそれに従って割りふっていくというやり方は暗合的強度の典型的な例になるでしょう．このような型でのモジュール化は，何もしないよりもまだ悪い結果をまねくことになります．

暗合的強度をもつモジュールは，プログラム内の活動範囲を個々のモジュール内にとどめるのではなく，むしろその反対のことを行っています．このタイプのモジュール内の各命令(ステートメント)は，関連性はあったとしてもそれは漠然としたものであり，逆に他のモジュール内の命令と密接に関連しあっているのです．

もし，プログラムに変更が生じた場合，プログラムがいくつかの暗合的強度モジュールに分割されていれば，その変更はすべてのモジュールに影響を及ぼすことになるでしょう．1つの変更が多くのモジュールに影響すればするほど，エラーの入り込む可能性が大きくなり，プログラムの信頼性をそこねることになります．

要するに，暗合的強度をもつモジュールは独立していません．他のモジュールとの関連性が強く，インターフェースも複雑になります．プログラムの保守性は低下し，再使用の可能性もほとんどありません．

図4.2に暗合的強度をもつモジュールの例を示しておきます．このモジュールで何をやろうとしているのか，簡単に説明することはできません．無理に説明しようとすれば，おそらく，このモジュールの手順，すなわち，論理を説明することになってしまうでしょう．

図4.2 暗号的強度の例

```
coin (x, y)
int  x, y, z ;
{
   z = x + y * 10 ;
   printf ("%5d %5d %5d\n", x, y, z) ;
   if (z > 100)
      z = 100 ;
   return (z) ;
}
```

たとえば，「yを10倍して，それにxを加えたものをzとし，zが100より大きくなれば，zを100にセットして，それをこのモジュールの出力として戻す，x，y，zの元の値は印刷する」といった説明になるはずです．

このようなモジュールを作る動機は，おそらく，このようなパターンがプログラム内に何度も出てくるとか，プログラムを何の考えもなしに，無造作にいくつかの部分に分断してしまったとかの場合でしょう．結果として考えられるのは，このモジュールで実行しようとしていることは，何か大きな機能のごく一部であり，その機能の残りの部分は，他のモジュール(複数個あるかもしれない)で実行されるだろうということです．

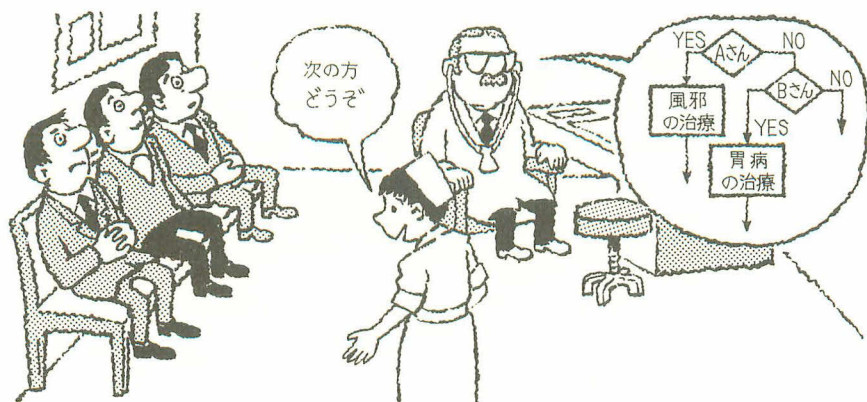
このような場合，その機能の変更は，このモジュールを含め，いくつかのモジュールの変更をうながすでしょう．また，このモジュールを他のプログラムが再利用する可能性はほとんどないと考えてさしつかえありません．

暗合的強度になってしまうようなモジュール化は，できるだけ避けるようにしなければなりません．

〔2〕論理的強度をもつモジュール

暗合的強度をもつモジュールは，通常の神経の持ち主ならば，まず作らないはずです．したがって，実際にはほとんど見かけません．

しかし，論理的強度をもつモジュールは，あちこちでよく見かけます．読者の皆さんのなかにも，この種のモジュールを作った経験をもっておられる方が多いものと思います．しかし強度の面からいえば，論理的強度をもつモジュールは，暗合的強度をもつモジュールについで良くないタイプに属します．「論理的強度をもつモジュール」とは，関連した

図4.3 論理的強度は町の内科医院

いくつかの機能を含み，そのうちの一つが呼出しモジュールによって指定され，実行されるモジュールのことです(図4.3)．論理的強度をもつモジュールが使用されるときは，必ず複数機能のうちどれを実行するのかの指示をパラメータで指定する必要があります．図2.1に示した好ましくないプログラム例は，じつはこのケースに該当します．ここで，もう一度このプログラムを見直してみましょう．見やすくするために，同じものを図4.4に再度示します．これは「大気の汚染度を測定する」プログラムなのですが，このプログラムでは，時間内の汚染度変化の様子をテーブルに記憶します(表4.2)．測定結果を貯えるテーブルに対していくつかの機能が必要になります．たとえば，テーブルの値をクリアするとか，テーブルにデータを追加したり逆に削除したり，あるいはテーブルのデータを探索参照したりするようなことが，必要になるでしょう．

このテーブルに関するいくつかの機能を1つのモジュール内にまとめ，実行させることは最近注目されているオブジェクト指向プログラミングにそったものであり，実用上いくつかの利点があります．たとえば，テーブルの仕様を変更したとき，変更がそのモジュールに限定できることです．これはソフトウェアの信頼性の面からは大変好ましいことです．一つの変更がいくつかのモジュールに波及すると，それだけ変更が大変になり，エラーにつながることは暗合的強度のところで考えました．したがって，現実にはこの種の論理的強度モジュールは変更の波及効果を小さくすることからよく作成されます．

図4.4 好ましくないプログラム

```
main()
  long ppm[60], pol;
  int   time, fcode, ecode;
  int   *ppol, *pecode;
{
  ppol=&pol;
  pecode=&ecode;
     ⋮
  fcode=0;
  table (fcode, time, *ppol, *pecode, ppm);
     ⋮
  fcode=1;
  table (fcode, time, *ppol, *pecode, ppm);
     ⋮
  fcode=2;
  table (fcode, time, *ppol, *pecode, ppm);
     ⋮
  fcode=3;
  table (fcode, time, *ppol, *pecode, ppm);
    ⋮
}
```

表4.2 汚染度テーブル

汚染度(ppm)	
81080	ppm〔0〕
92530	ppm〔1〕
81850	
⋮	⋮
83000	
82850	ppm〔59〕

▶論理的強度の欠点

しかし，モジュール強度的にみれば，先述したように，つぎのような大きな問題点を有しています．

(1) 複数機能に対し単一インターフェースをもつ．

(2) 変更が高価になりがちである．

(1)の問題は，インターフェースを必要以上にこみ入ったものにし，わかりにくいものにしています．

表4.3は，大気汚染度テーブル操作モジュールtableのインターフェース仕様をまとめたものです．5種類のインターフェース・データがモジュール(関数)tableの引数としてうけわたしされます．プログラム内では，引数fcode，time，pol，ecode，ppmがインターフェース・データにそれぞれ該当します．先にも述べたように，インターフェース・データは機能コード(fcode)によって指定された機能によって意味が異なってきます．テーブルのクリア機能に対しては，インターフェース・データ2，3，4は意味をもちません．すなわち，ダミーになります．

また，インターフェース・データ3の汚染度(pol)は，追加機能に対しては入力，探索機

表4.3 大気汚染度テーブル操作モジュールのインターフェース

インターフェース・データ					
1		2	3	4	5
機能コード		時　間	汚染度	エラー・コード	テーブル
0	クリア	N・A	N・A	N・A	入　力
1	追　加	入　力	入　力	出　力	出　力
2	削　除	入　力	N・A	出　力	入　力
3	探　索	入　力	出　力	出　力	入　力

N.A：Not Available

能に対しては出力というように変わってきます．エラー・コード(ecode)も出力であることは変わりませんが，機能ごとにその値の意味するところは変ります．たとえば，エラー・コードの値が1のときは，追加機能では「テーブル・オーバフロー」，削除，探索機能では「該当項目なし」といったことになるでしょう．これらの意味を間違えずに正しく理解することは，それほどなまやさしいことではありません．

論理的強度モジュールを作る動機は，変更を局所化することにあることはすでに述べましたが，これも変更の内容によって話が異ってきます．(2)の問題は，このことを指摘しています．複数機能のうちのどれか一つの機能の変更が必要になったとしましょう．それが引数の数を増す必要があるような変更であれば，その変更と関係のない機能を使用しているモジュールにも波及してしまいます．たとえば，fcode＝2，すなわち，テーブルからデータを削除する場合に，状況により，データをテーブルから完全に削除してしまうか，あるいは，削除フラッグだけをつけるにとどめ，データはテーブル内に残しておくかのどちらかに分けたくなったとします．その場合，どちらにするかを指示するために，引数をあらたに1個追加したとしましょう．いまや，モジュール tableの引数は5個から6個になってしまいます．そして，これは，本来関係のない fcode＝0，1，3の機能にも影響を及ぼします．なぜなら，モジュール tableを呼出すときは，その機能の如何にかかわらず，引数を5個から6個に指定変えしなければならないからです．このモジュールを呼び出しているプログラムが10個も20個もあるとすれば，考えるだけでいやになってしまいますね．

このほかに，論理的強度モジュールは，機能を指定するための機能コードを必ず持つがゆえに，機能コードの値をチェックし，それが正しいかどうかを調べる論理が必要になっ

てきます．そして，もし値がおかしければエラー・メッセージを打出し，おかしいことを知らせる必要があります．たとえば，上の例では，fcodeが0～3以外の値であればエラー・メッセージでエラーを知らせる必要があります．これらの処理は，機能コードがあるがために必要になるものであり，機能コードがなければ，必要のないものです．モジュールを単一機能を実行するように設計すれば，機能コードは当然のことながら不要になり，これらの処理も不要になります．

このように，論理的強度をもつモジュールは良い点ももっていますが，大きな欠点ももっています．良い点を生かす方法は別にもありますので，なるべくならこの種のモジュールはさけたほうが賢明です．

4.2　強度の強くなるモジュール化について考えてみよう

〔1〕機能的強度をもつモジュール

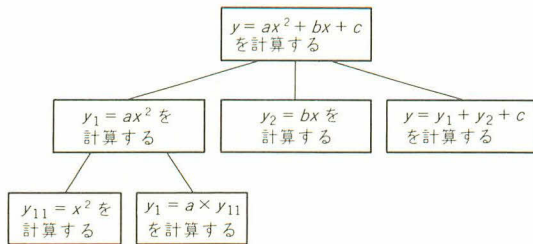
暗合的強度，論理的強度が強度的にあまり好ましくなかったのに対し，機能的強度は強度としては一番強く，モジュール化にさいしては，できるだけこの型にもっていくのが好ましいと言えます．

「機能的強度をもつモジュール」とは，単一の機能を実行するモジュールです(図4.5)．図4.6は機能的強度をもつモジュールの例で，「配列を合計する」という単一機能を遂行するモジュールです．配列の合計を求めるために，モジュール内のすべての命令がお互い

図4.5　機能的強度は夫婦の絆

図4.6　機能的強度の例

```
matadd (a, n)
int a [ ], n;
{
    int i, sum;
    sum = 0;
    for (i = 0; i < n; i + +)
      sum + = a [i];
    return (sum);
}
```

図4.7　機能展開の例

に関連しています．命令のどれ1つも欠かせません．このようなケースでは一部の命令だけを変更する可能性は大変少なくなります．また，たとえ変更するにしても，その機能に対する変更であり，そのモジュールだけで処理でき，他のモジュールへの影響度合は小さくなります．このような理由により，強度が強いと判定できるわけです．

▶単一機能とは何か

ここで，注意する点は単一機能とは何かということです．それがはっきりしないと，モジュール化にさいして機能的強度を持たすことが難かしくなり，個人差が出てしまいます．

機能とは，簡単にいえば入力を処理して出力に変換することです．数学的表現をするなら，$y=f(x)$です．その意味では，xをたんにyにおきかえる$y=x$も機能であり，2次方程式$y=ax^2+bx+c$も機能になります．とくに，2次方程式の場合は，それ自体1つの機能でありながら，それをいくつかの部分機能に分解できることを示しています．

たとえば，2次方程式を計算する過程$y_1=ax^2$，$y_2=bx$も1つの機能とみなすことができます．なぜなら，aとxを入力にしてy_1という出力に変換しています．そして最後に$y=y_1+y_2+c$を計算して目的を達成します．このことを整理すると図4.7のようになります．この図から，機能は階層的に展開できること，それを行うことで問題が理解しやすくなることがわかっていただけるでしょう．じつは，このことはすでに図3.7～図3.8で検討した「病院の入院患者を監視する」例でもみてきたことなのです．

さて，それでは単一機能とは何を意味するのでしょうか．図4.7で示されているすべての機能は単一機能です．2次方程式や患者の監視の例にみられるように，たいていの機能はいくつかのより低位の部分機能から構成されます．それらの複数機能をまとめて1つの機能ということができます．

2次方程式を計算する機能は，3つの単一機能「$y_1 = ax^2$を計算する」，「$y_2 = bx$ を計算する」，「$y = y_1 + y_2 + c$ を計算する」からなりたつ単一機能です．

また，「患者を監視する」は3つの単一機能「患者の諸要因を得る」，「危険要因をみつける」，「危険要因を看護婦に知らせる」からなりたつ単一機能です．

機能的強度をもつモジュールは単一機能を実行するモジュールです．「2次方程式を計算する」あるいは「患者を監視する」を1つのモジュールで実行させるときは，そのモジュールは機能的強度をもちます．

▶機能的強度をもたない例

もう1つ別の例を考えてみましょう．$y_1 = 5x + 3$ を計算する，$y_2 = 3x^2 + 6x + 9$ を計算するという2つの単一機能があったとします．この2つの機能がとくに関連があるわけでないとき，これらをまとめて一つの上位機能として表現できるでしょうか．むづかしいですね．せいぜい「$y_1 = 5x + 3$ を計算し，つぎに $y_2 = 3x^2 + 6x + 9$ を計算する」くらいにしか表現できません．これを1つのモジュールにまとめたとき，そのモジュールは機能的強度をもつとは考えません，ソフトウェアの仕様書のなかに，y_1を計算し，つぎにy_2を計算することを指示してあれば，これはこのあと説明する手順的強度になります．とくに指示がなければ，暗合的強度とみるほかないでしょう．

分割したモジュールが，機能的強度になっているかどうかをごく簡単に判定する方法があります．それは，そのモジュールで実行しようとしていることを説明するときに，1つの目的語と1つの述語で表現できるかどうかを調べればよいのです．ですから，そのときそのモジュールは機能的強度モジュールをもつとみなしてよいでしょう．

〔2〕情報的強度をもつモジュール

構造化設計では，モジュールを機能的強度にすることが望ましいのですが，その理由は変更が生じたとき，その影響範囲を局所化できること，再使用性が高まることなどでした．変更を1つのモジュール内に留まらせるという意味では，情報的強度も望ましい型ということができます．

「情報的強度をもつモジュール」とは，ある特定のデータ構造とかテーブルに対する複数の操作を1つのモジュール内に隔離したモジュールのことです(図4.8)．このことは情報隠匿の概念と呼ばれています．情報隠匿によってデータ構造とかテーブルに関する重要な変更を他のモジュールに波及させることなく行うことができます．

図4.8 情報的強度は担当部長

▶情報的強度とインターフェース設計

先述の大気汚染問題の例で，テーブルに対する複数の操作を１つのモジュールにまとめた論理的強度をもつモジュールを紹介しました．

すでにおわかりの通り，論理的強度モジュールには，あまり好ましくないいくつかの特性がありましたが，情報隠匿の概念からいえば，うまいまとめ方をしているわけです．

情報的強度は論理的強度と情報隠匿の概念では共通しています．しかし，決定的な違いがあります．それはインターフェースの設計です．論理的強度をもつモジュールでは，複数の機能に対して１つのインターフェースで操作します．情報的強度をもつモジュールはそれぞれの機能に対して固有のインターフェースを設計します．すなわち，機能ごとにインターフェースの持ち方をかえます．それによってダミーの引数や，機能が異なると同じ引数でも意味がちがってくるといったわずらわしさをなくすことができます．

具体例でみてみましょう．図4.9は情報的強度モジュールの例です．この例は，図4.4の論理的強度モジュール table を情報的強度モジュールに変換したものです．論理的強度モジュール table は，テーブルに関する４つの機能，すなわち，クリア，追加，削除，探索を１つのモジュールで行おうとするものでした．それに対して，図4.9の例は，内容的には table モジュールと同じことを行おうとしています．ただ，入口点が４つあり，それぞれに対して異なる４つの名前がつけられています．TABLEC，TABLEA，TABLED，TABLES の４つです．

これらの入口点は，クリア，追加，削除，探索の４つの機能に対応しています．また，

図4.9 情報的強度の例(PL/I 言語)

```
TABLEC : PROCEDURE ;
          モジュール全体のデータ宣言
          テーブル・クリア機能に固有のデータ宣言
          テーブル・クリア機能に対するコード
          ⋮   ⋮
          RETURN ;
TABLEA : ENTRY (TIME, POL) ;
          データ追加機能に固有のデータ宣言
          データ追加機能に対するコード
                ⋮
          RETURN ;
TABLED : ENTRY (TIME) ;
          データ削除機能に固有のデータ宣言
          データ削除機能に対するコード
                ⋮
          RETURN ;
TABLES : ENTRY (TIME) ;
          データ探索機能に固有のデータ宣言
          データ探索機能に対するコード
                ⋮
          RETURN ;
```

それぞれの機能を実行するときに必要になる引数データが，それぞれの入口点に指定されています．機能ごとに引数が異なっている点に注目して下さい．

図4.9のこのモジュールは，物理的には1つのまとまったモジュールです．しかし，いま説明したように，それぞれの機能ごとに異なった入口点をもち，必要な機能ごとにその入口点から実行を開始します．たとえば，テーブルにデータを追加するときは，入口点TABLEAから実行を開始します．そして，RETURN命令により呼び出しモジュールに戻ります．

それぞれの機能に対する入口点と出口点は，他の機能のそれとは独立していますので，その意味では，情報的強度モジュールは，物理的には1つのモジュールでも，論理的には機能の数だけの独立したモジュールから構成されていると考えてよいでしょう．

モジュールをこのように設計しておけば，それを使用するときは，図4.10の形式になってくるでしょう．それぞれの機能に対応する異なる入口点名と引数が指示されていることに注意して下さい．

図4.10 情報的強度モジュールの呼び出し

```
static int pol_tab[2] [1440], opol, ecode;
main ( )
{
  int time, pol ;
   ⋮
  tablec ;
   ⋮
  tablea (time, pol) ;
   ⋮
  tabled (time) ;
   ⋮
  tables (time) ;
   ⋮
}
```

このことから，モジュールを情報的強度に設計することで，情報隠匿の概念をいかしながら論理的強度がもっていたいくつかの問題点が解消されることがわかります．すなわち，機能コードに対する引数が必要でなくなり，それにともなうわずらわしさが解消します．また，各機能ごとに固有の引数を指定できることにより，機能ごとに同じ引数の解釈が異なるといったこともなくなってしまいます．そして，機能固有の変更が，他の無関係な機能に影響を及ぼすといったこともなくなるでしょう．

ただ，このような多重入口点をもつモジュールを作る場合は，プログラミング言語の制約をうけることになります．残念ながら，現在，製品として世に出ているプログラミング言語のなかで，多重入口点機能を使用できるのは，それほど多くありません．図4.9の例は，PL/I 言語を想定して作られていますが，PL/I が使用できるマイコンは，ほとんどないのが現状です．

言語の制約から情報的強度モジュールが作れないときは，無理をせず，機能ごとに，単一機能を実行する別個の機能的強度モジュールを作るよう心がけましょう．

C 言語では，多重入口点機能はありませんが，機能ごとに別関数を作成し，それらをまとめて１つのファイルとしてコンパイルすれば，結果として，多重入口点機能と同等のことを行うことができます．

C の上位言語である C＋＋では，情報的強度がねらいとしている情報隠匿の概念を具現化するためにはクラスを用いることもできます．たとえば，次のようにクラスを定義します．

```
class tableop
   {
      long ppm[60];
    public:
      void tablec(long＊);
      void tablea(int, long, long＊);
      void tabled(int, long＊);
      void tables(int, long＊, long＊);
   };
```

この定義では，情報隠匿の対象になっているオブジェクト，ppm テーブルはクラス内のメンバ関数だけが操作でき，プログラムの他の部分からは操作できないようにしています．しかし，ここで定義された4つのメンバ関数 tablec，tablea，tabled，tables はプログラムのどの部分でも使用できます．

C＋＋では，クラスは利用者が目的に応じて定義する1つのタイプ(long とか int と同じ)ですから，クラスを使用するときは

```
tableop ppmtbl;
```

のように宣言する必要があります．そして，クラス内のメンバ関数をプログラムのどこかで使用するときは

```
        ⋮
ppmtbl.tablec(ppm);
        ⋮
ppmtbl.tablea(time, pol, ppm);
        ⋮
ppmtbl.tabled(time, ppm);
        ⋮
```

のように使用します．

このような形でクラスを定義しておけば，オブジェクトであるテーブル ppm はクラス内だけで処理できることになり，もしテーブルに変更が生じてもクラス内だけで処理することができ，変更の波及効果を局所化できることになります．

なお，メンバ関数を定義する(クラス内では宣言しているだけ)には，次のように行います．

```
void tableop :: tablec(long* ppm)
{
        :
}
```

ここで，tableop はクラス名であり，tablec はメンバ関数名です．クラス名とメンバ関数名の間に :: を必ず入れるようにします．

このように，オブジェクト単位にクラスを定義し，プログラムで使用していくのがオブジェクト指向プログラミングであり，情報的強度モジュールはまさにこの方向にそったものであるということができます．

4.3　その他の強度をもつモジュールについて考えてみよう

いままでに説明した強度の他に，まだいくつかの強度が考えられます．それは，時間的強度，手順的強度，連絡的強度といったものです．これらの強度は，強度的には中間に属しますが，利点より欠点の方が目につきます．

その意味では，構造化設計を行うさいには，特に強い理由がない限り，これらの強度になるように設計するのはなるべく避けた方が無難です．

以下，3 つの強度について簡単な例を通してみてみましょう．

図4.11 は A, B 2 つのタイプのトランザクションを読み取り，それを処理するプログラムの処理手順の大枠を示したものです．

この図をもとに，このプログラムをいくつかのモジュールに分割します．いま，図4.11 の太線で囲んだ 4 つのモジュールに分割したものとします．

モジュール INIT は初期設定を行います．ファイルのオープンであるとか，合計域，テーブルのクリア作業などがこのモジュールの役目になります．これらの作業を最初にまとめてやってしまうことは，プログラムならだれでも考えることだと思います．

モジュール MAINPROC は，全トランザクションを処理します．もちろん，このプログラマの中心になる部分です．この MAINPROC は，トランザクションのタイプを調べ，タイプごとに異なった処理を行う部分を RECPROC という子モジュールとして分割し，必要に応じて呼び出すようにしています．4 つ目のモジュール TERM は終了処理モジュールです．最終合計の出力などがその作業内容になるでしょう．

この 4 つのモジュール構造を構造化チャートで表現すると図4.12 のようになります．

図4.11 2種類のトランザクションを処理するプログラムの大枠手順

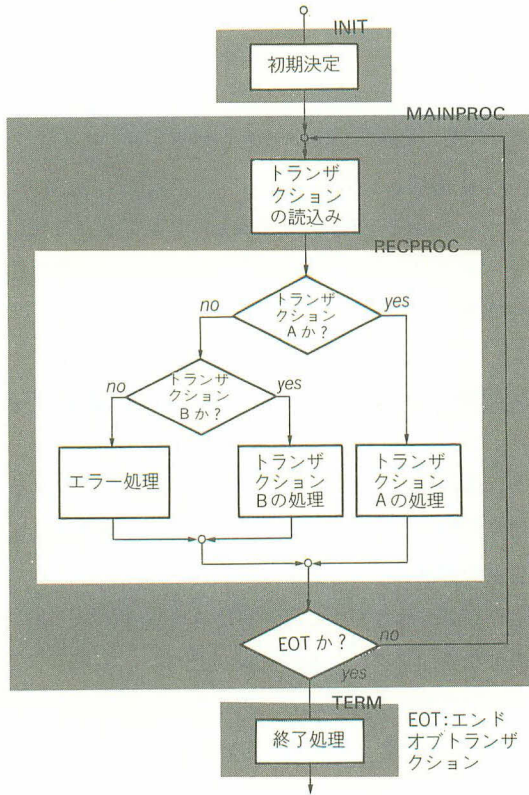
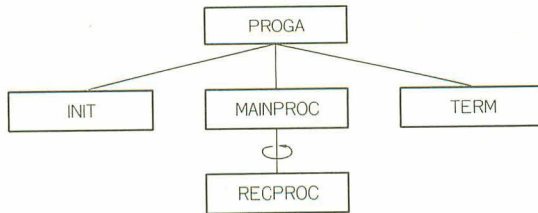



図4.12 図4.11のプログラム・モジュール構造

さて，このように分割した4つのモジュールを強度的にみてみましょう．

〔1〕時間的強度をもつモジュール

モジュールINITについて考えてみます．このモジュールの特徴は，“プログラム実行開始の段階で行っておかなければならないことをすべてまとめて行っている”ことです．その意味では，このモジュール内の命令は時間的に強い関連をもっています．このようなモジュールを「時間的強度をもつモジュール」とよびます．

この観点からみれば，モジュールTERMも時間的強度をもつことが容易に理解できるはずです．時間的強度をもつモジュールは，特定の時点で行うべきことをすべてまとめて行うため，プログラマにとっては考えやすい面があることは事実であり，現実にもよくこの型のモジュールが作成されます．

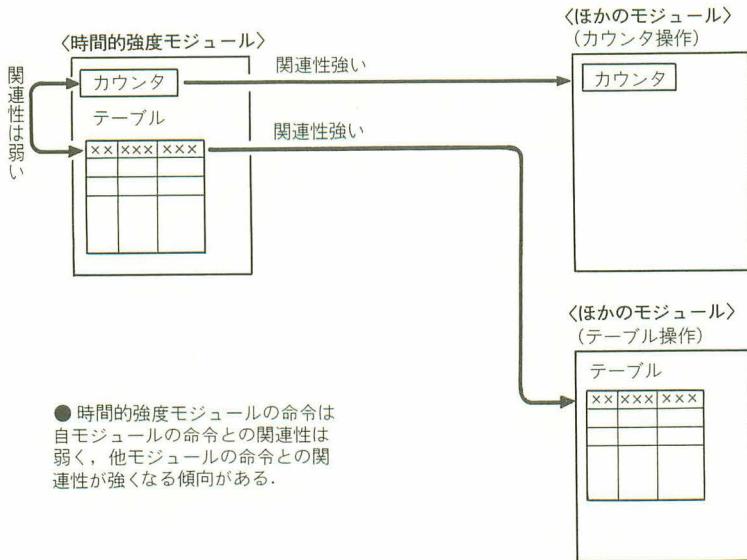
しかし，時間的強度モジュールは大きな問題点をもっといます．それは，このモジュール内の命令が機能的にみれば，互いにあまり強い関連をもっていないことです．逆に，ほかのモジュール内にある命令と強い関連をもつ可能性があります．カウンタのリセット，テーブルのクリアなどは互いに機能的に関連があるわけではなく，たまたまプログラムの実行開始時点で行うという共通点をもつがゆえに1つのモジュール内に入っているにすぎません．むしろ，カウンタはカウンタを処理するほかのモジュール内の命令と強い関連をもちますし，テーブルもテーブルを処理するほかのモジュール内の命令と強い関連をもちます．

このことは，カウンタやテーブルについて考えるとき，INITだけでなく，ほかのモジュールのことも考えなくてはならないことを示唆しています．もし，カウンタのけた数やテーブルの大きさの変更が必要になったとき，関連するモジュールすべてに影響します(図4.13)．

この意味で，時間的強度は独立性が高いとはいえず，分割の形としてはあまり好ましくありません．

〔2〕手順的強度をもつモジュール

次に，モジュールMAINPROCとRECPROCについて考えてみます．MAINPROCは，図4.11の問題解決の大枠の手順を表現した流れ図の一部分をまとめてモジュールにしたものです．また，RECPROCはMIANPROCに含まれた手順の一部をまとめてモジュールにしたものです．

図4.13　時間的強度モジュール

このことは，MAINPROC と RECPROC は親子関係(RECPROC が MAINPROC に従属している)であることを表しています．図4.12 のモジュール構造はこのことを示しています．

MAINPROC，RECPROC のように，問題解決のための手順をもとに，その手順の一部をモジュールとしてまとめたものを「手順的強度をもつモジュール」とよびます．

手順的強度の特徴についてみてみましょう．手順的強度をもつモジュールは，問題解決のための手順に基づいてモジュール化したものですから，モジュール内の命令も当然問題解決手順的に関連しており，それなりに関連性の強さをもっています．少なくとも，先述の時間的強度よりは強度が強いとみなせます．

しかし，手順的であるということは，機能的であることとは本質的に違います．１つのモジュール内に収められた手順は，機能的にみれば，単一機能のうちの一部だけを実行するかもしれません．あるいは，逆に複数の機能を含んでいるかもしれません．

RECPROC をみても，そのなかにはトランザクション A と B の処理，エラー処理などを含んでいます．これらはそれぞれ１つの機能とみなすことができるので，RECPROC は複数個の機能を含んでいるとみなせます．

これは RECPROC を作成するとき，A, B 両方の処理とエラー処理を考える必要がある

ということであり，同時に考慮しなければならない要素の数がそれだけ多くなることを意味しています．また，Aに対する変更が生じたとき，Aの処理とそれ以外の処理とを見分けながら手当てしていかなければならず．修正作業を困難にします．

前節でみたように，最適なモジュール化は単一機能を1つのモジュールにまとめることであり，本質的に手順的強度モジュールはこの原則と一致しません．

機能と手順は，本質的には違うものであることを認識しておく必要があります．機能は何(what)を行うかを表すものであり，手順は機能を実行する方法(how)を示します．先に機能と論理を区別した理由は，まさにそこにあります．

プログラム設計では，まず，そのプログラムで行うべきことを明らかにし，次にそれを行うための手順を考えていきます．その大きな理由は，機能と手順を明確に区別せず，最初から手順指向で設計すると，そのプログラムで何を行うべきかという最も大切な点がぼけてしまうからです．

モジュール化は，プログラムの機能を明確にするねらいをもっています．その意味で，手順的強度はモジュール化の本質的な意図にそぐわないのです．

モジュールが手順的強度になってしまうのは，流れ図のような手順を示すものをもとにモジュール化することに起因します．流れ図は機能的にモジュール化することに対して何も伝えてくれないことを知っておく必要があります．

時間的強度と手順的強度は，モジュールの独立性の面からみれば，あまり望ましい形でないことは理解していただけたと思います．

それでは，時間的強度，手順的強度がもっている問題点を解消し，よりよい機能的なモジュール化を図るにはどうしたらよいでしょうか．

図4.14 図4.11の問題をDFDで表現したもの

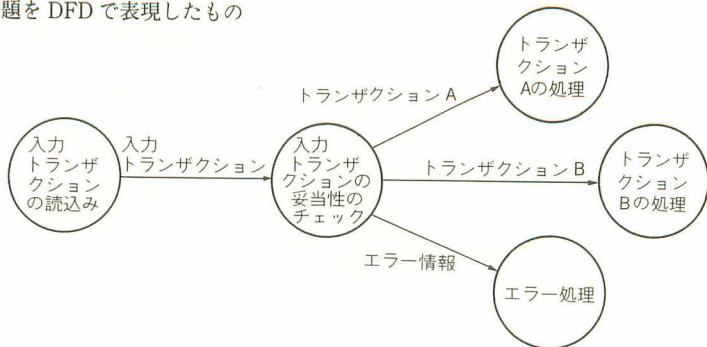
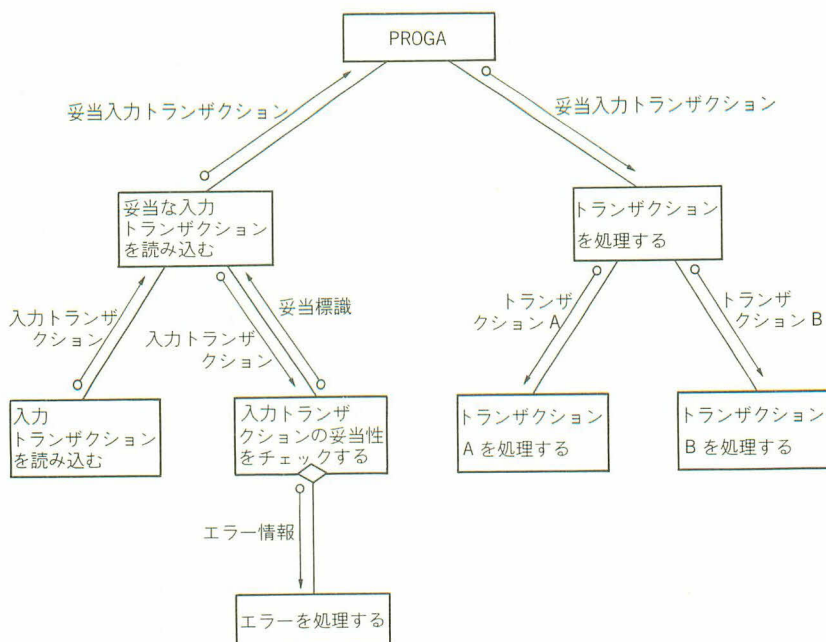



図4.15　DFDをもとに作成したモジュール構造

▶ DFD(Data Flow Diagram)

1つの方法は，モジュール化を考えるさいに，手順指向の流れ図でなく，機能指向のデータ・フロー・ダイアグラム(以下，DFDと略す)を用いて問題を記述してみることです．図4.11の流れ図で示した問題をDFDで記述し直したものを図4.14に示します．

図4.14で，矢印はデータの流れを示しています．また，丸印は機能(処理)を示しています．DFDでは，機能は入力データを出力データに変換する処理としてとらえられています．DFDによって，問題内のデータの流れとその変換過程としての機能を明確に表現することができます．手順を表現する流れ図との違いをよく吟味してください．DFDには，初期設定，終了処理，トランザクションのタイプを判別する選択手順，EOFを条件とする繰返し手順といったものは一切表現されていません．DFDで表現されているのは，データの流れと機能であり，それによってプログラムで行うべきことが明確に理解できます．

DFDをもとにモジュール化を図り，構造的に表現したものが図4.15です．図4.15に表

現されているモジュールは，それぞれ単一機能を実行するようになっています．先述したように，最適なモジュール化は，モジュールに機能的強度をもたせることです．

機能的強度モジュールは，問題解決や変更をそのモジュールだけに局所化することを可能にします．図4.15で，トランザクションＡの変更は「トランザクションＡの処理」モジュールだけで考えることができ，ほかのモジュールへの波及効果を少なくします．ほかへの影響が少ないということは，それだけモジュール強度が強いことを意味しています．

〔３〕連絡的強度をもつモジュール

最後に，連絡的強度について説明しておきましょう．連絡的強度は，本質的には，手順的強度と同じです．すなわち，問題解決のための手順を基本としてモジュール化していきます．したがって，１つのモジュール内に複数の機能を含んだり，単一機能の一部を含んだりします．

連絡的強度が手順的強度と異なる点は，モジュール内の機能がデータの使用に関して互いに関連し合っていることです．たとえば，モジュール内のすべての機能が同じ情報を参照しているような場合です．最初の機能の出力が２番目の機能の入力になり，２番目の機能の出力が３番目の機能の入力として使われるといったものがこれに該当します．たとえば，入力トランザクションを読み取り，その妥当性をチェックし，妥当であればデータベースに蓄える機能を１つのモジュールにまとめたものは連絡的強度モジュールとみなすことができます(図4.16)．

連絡的強度は，同じデータに関して関連し合った手順ということで，単なる手順的強度より強度的には強くなります．

図4.16 連絡的強度モジュールの例

```
入力ｘを読取る
ｘの形式チェックを行う
IF エラー THEN
    エラー・メッセージを書出す
ELSE
    ｘを入力ファイルに書出す
ｘを合計域に加える
```

コラムC

C++言語

C++言語は，C言語の後継言語として注目されています．C++は，C言語のスーパーセットとして米国ベル研究所で開発されました．

C言語がもっている効率のよさや可搬性は，C++でもそのまま生かされています．C++言語では，C言語で記述できることはすべて記述できますし，C言語しかもっていない新たな機能ももっています．そのなかでの代表的な機能としてクラス機能があります．クラス機能によって，設計上の重要な概念である情報隠匿の概念が実現できることになります．

情報隠匿の概念は，最近注目されているオブジェクト指向プログラミングでもいかされています．

プログラムの構造化設計における1つの重要な目標は，モジュール化に際して，機能的強度モジュールか情報的強度モジュールのどちらかになるようにすることです．これらの強度をもつモジュールは，モジュールの独立性を最大にし，プログラムのエラーを減少させます．また，変更や追加をやりやすくし，プログラムの拡張性を高めます．さらに，このプログラムあるいは今後作成されるプログラムで，モジュールの再使用の可能性を高めることになります．

コラムD

情報隠匿の概念とクラス機能

C＋＋言語は，設計時の情報隠匿の概念を実現させるためにクラス機能をもっています．

情報隠匿の概念とは，局所的にしか使わない情報は，関係のない他の部分から操作できないように，隠してしまい，誤まった操作から情報を守ります．これはモジュール設計時にブラック・ボックス化を可能にし，モジュールの独立性を高めます．結果として，変更を局所化し，保守作業をやりやすくします．また，データのセキュリティを高めることも可能にします．

クラス機能は，この情報隠匿の概念を次のような形で実現させます．

```
class datastruct
  {
   int x, y, z;
       :
   public;
     int s, t, u;
       :
  };
```

この例で，

```
class datastruct
```

はクラスの宣言部の最初の行であり，class はクラスを宣言するためのキー用語，datastruct は構造体名です．

この行から public; の部分までが非公開部(Private　Part)であり，残りの部分が公開部(Public Part)です．

非公開部のデータ(この例では x，y，z)は，メンバ関数しかアクセスできません．一方，公開数のデータ(s, t, u)はどこからでもアクセスできます．

5．モジュール間結合度を弱くすることを考えよう

モジュール強度は，１つのジュール内部の関連性についての尺度であり，関連性が強いほどそのモジュールの強度は強く，独立性が高いと考えるものでした．モジュールの独立性に関するもう１つの尺度は，**モジュール間結合度**です(図5.1)．

モジュール間結合度は，モジュール間の関連性について考察するものです．モジュール間の関連性が弱いほど，個々のモジュールの独立性が強くなるわけですから，結合度が弱いほど設計としては好ましい結果になります(図5.2)．

モジュール間の結合の仕方としては，**表**5.1のような六つのタイプが考えられます．**表**5.1では，結合度の強いものから弱いものへと上から下への順でならべるように設計することが望ましいのです．結合度の強弱は，モジュールのエラー傾向，テストの困難さ，モジュールの再使用性，保守性などの要因に対して相対的にどのような効果をもっているかで判定します．

図5.1　モジュール独立性の尺度

モジュール強度：
特定モジュール内の構成要素間の関連性の強さを表す尺度

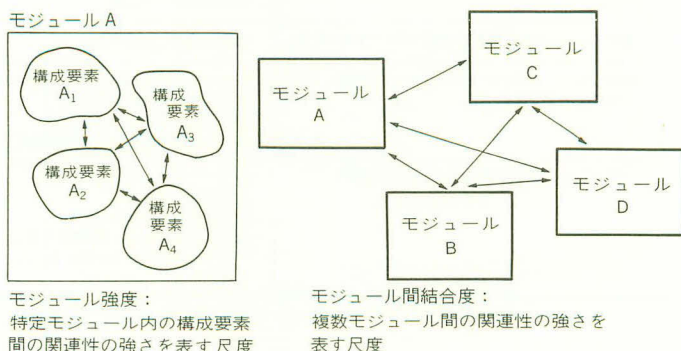
モジュール間結合度：
複数モジュール間の関連性の強さを表す尺度

図5.2　結合度にもいろいろあって…

強い結合度

弱い結合度

表5.1　モジュール結合度一覧

（強い ↔ 弱い）

種類	定　義	特　徴
内　容	・他のモジュールの内部を直接参照する． ・CALLではなく直接ブランチ (アセンブリ言語)	・他のモジュールの変更に影響される．
共　通	・いくつかのモジュールが同じ大域的データ項目（異種）を参照する．	・プログラムが読みにくい（パラメータで指定されない）． ・他のモジュールの変更の影響をうける． ・あとから利用しにくい． (共通域データのSAVE/RESTORE) ・不必要なデータ定義のわずらわしさと不用意な使用．
外　部	・いくつかのモジュールが同じ大域的データ項目（同種）を参照する．	・プログラムが読みにくい． ・あとから利用しにくい． ・不必要なデータ定義．
制　御	・他モジュールの論理を制御するパラメータをわたす．	・論理的強度になりやすい．
スタンプ	・使用しない一部のデータを含むデータ構造をパラメータによって受渡しする．	・レコード内の無関係な変更に影響される． ・必要以上にデータをさらす． ・ブラック・ボックス化が可能． ・読みにくさや利用しにくさは解消．
データ	・パラメータによる同種のデータ項目の受渡し．	・ブラック・ボックス化が可能． ・再使用性が大きい． ・無関係なデータをさらさない． ・結合の範囲が小さい(関係する2モジュールのみ) ・パラメータ・リストが長くなったときのわずらわしさがある．

結合度について考えるときは，それがモジュール間の関連性を記述するものなので，複数のモジュールが対象になります．モジュールＡがデータ結合であるというようないい方は意味ありません．モジュールＡとＢがデータ結合であるといういい方になります．

5.1　結合度が強くなってしまうモジュール化は，どのような時におきるのだろうか

〔1〕内容結合になるモジュール

モジュール強度と同様に，もっともよくないケースを考えることで，結合度の面からやってはいけないことを理解し，ひいては望ましい型を考察してみたいと思います．「内容結合」は，２つのモジュールで一つが他の内部を直接参照したり，一部のルーチンを共有したりする結合の型です(図5.3)．この場合，一方のモジュールの変更が他のモジュールに影響をおよぼすことになります．変更の波及効果が大きくなればなるほど，品質面での保証が難しくなり，ソフトウェアの信頼性に問題が出てきます．アセンブリ言語などで，ブランチ命令で直接他のモジュールの部分へ制御を移すことによって，このような結合の型を発生させることができますが，さいわいなことに，FORTRANやＣ言語を通常の方法で使用してプログラミングをしているかぎり，内容結合が発生することは，ほとんどありません．

〔2〕共通結合になるモジュール

図5.3　内容結合の概念

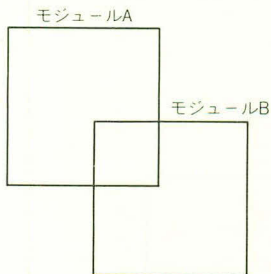
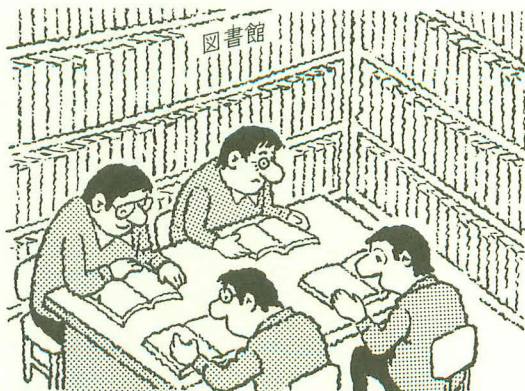



図5.4　共通結合は図書館のようなもの

「共通結合」は，結合度の強さについては内容結合についで強く，設計上あまり好ましい型ではありません．しかし，現実にはソフトウェアがこの型の結合度によって作られることはよくあります．強度における論理的強度と同様に，それを必要とする強い動機が存在するからです．

共通結合は，共通域にあるデータを参照する複数のモジュール間で発生する結合の型です(図5.4)．図5.5はその例を示しています．この例では，共通域に，X，Y，Zの3つのデータ構造を有しています．データ構造は，異種のいくつかのデータが集まって論理的な1つのグループ(レコード)を形成したものと考えて下さい．

この共通域のデータを3つのモジュールA, B, Cが共通しています．ただ，AはX, Yだけを使用し，Zは使用しません．一方，BはY, Zを使用し，Xは未使用，CはX, Zを使用し，Yを使用しません．こういった場合でも，ほとんどの言語では，モジュールA, B, Cに共通域のすべてのデータ(使用，未使用にかかわらず)を宣言する必要があります．共通結合という名前はFORTRANのCOMMON命令にあやかってつけられたものです．COMMON命令は，データ集合を特定の共通域に貯え，そこにあるデータを複数のモジュールで参照できるようにする命令です．図5.6はその例を示しています．この例で，モジュールMODA，MODB，MODCはCOMMON域にあるデータX, Y, Zを介して共通結合しています．

C言語では，複数個の外部変換を介して結合しているモジュールは共通結合になります．たとえば，図5.7のようなプログラムを考えてみましょう．

このプログラムで，x, y, zは外部変数であり，すべてのモジュール(関数)で広域的にアクセスできます．外部変換を介して結合しているモジュールは共通結合です．このプログラムでは，main，mod1，mod2は共通結合になります．

図5.5　共通結合の概念

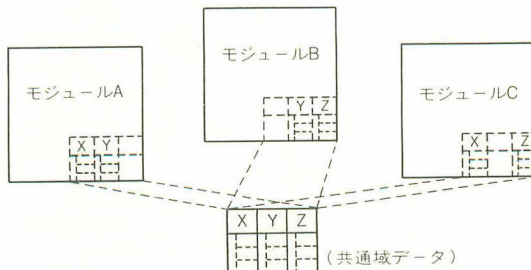



図5.6　共通結合の FORTRAN モジュール

MAIN

```
COMMON  X, Y, Z
  ⋮
CALL  MODA (R1, R2)
  ⋮
CALL  MODB (R3, R4)
  ⋮
CALL  MODC (R5, R6)
  ⋮
END
```

MODA

```
SUBROUTINE  MODA (R1, R2)
COMMON  X, Y, Z
  ⋮
R1=X+Y
  ⋮
R2=X−Y
  ⋮
RETURN
END
```

MODB

```
SUBROUTINE  MODB (R3, R4)
COMMON  X, Y, Z
  ⋮
R3=Y+Z
  ⋮
R4=Y−Z
  ⋮
RETURN
END
```

MODC

```
SUBROUTINE  MODC (R5, R6)
COMMON  X, Y, Z
  ⋮
R5=X+Z
  ⋮
R6=X−Z
  ⋮
RETURN
END
```

図5.7　C の外部変数は外部結合
となることもある

```
int x, y, z ;
main( )
{
  extern int x, y, z ;
    ⋮
}
mod1( )
{
  extern int x, y, z ;
    ⋮
}
mod2( )
{
  extern int x, y, z ;
    ⋮
}
```

▶共通結合のデメリット

さて，共通結合はなぜ内容結合についで結合度が高いとみなされるのでしょうか．それは，一般的には，つぎのような理由によります．

(1) 共通域のデータの一部が修正されたとき，共通域参照の全モジュールが影響をうける．

(2) 共通域のデータは，一部のデータを使用しないばあいでも，モジュール内ですべて宣言しておかなければならない(特にFORTRANの場合)．これは，モジュール内に本来不必要なデータをもちこむことになり，取扱いをわずらわしくし，また，まちがった使用を発生させる可能性をもっている．

(3) 共通域のデータは，モジュール間のインターフェースに現れないので，モジュールの読みやすさをそこねる原因になる．

(1) に関しては，図5.5で，モジュールAの都合でデータXの長さを変更したとき，本来その変更に無関係であるモジュールB，Cがその変更の影響をうけ，少なくともコンパイルのし直しが必要になります．まずいばあいには，B，CはAの変更により，共通域のデータを使えなくなったり，変更したのを知らずにそのまま使用して，思わぬトラブルにまきこまれてしまうこともあります．共通域のデータを数百個ものモジュールで参照しているプログラムも世の中に存在します．共通域のデータの一部の変更が，このようなプログラムではどんな意味をもつのか，ちょっと考えただけでもぞっとしますね．ソフトウェアの信頼性の面からみて，これひとつとっても，共通結合があまり好ましくないことがわかっていただけるでしょう．

(2) の問題に関しても，本来不必要なデータをもちこむことは，ソフトウェアをわずらわしくし，データの安全保護の面からも好ましい状態ではありません．

これらの問題点は，図5.6に示したFORTRANモジュールに対してはそのまま当てはまります．しかし，C言語の場合はやや趣が異なります．

Cでは図5.7のように，FORTRANのようなCOMMON命令は使用しません．共通データを使用する各モジュール内でCOMMONに該当するような宣言もしません．各モジュールで共通のデータかどうかは，プログラムの文脈上から決まることであり，特別の宣言はありません．(図5.7では外部変数であることを明確にするため，あえて宣言してあります)．これは少なくとも，COMMONのように，各モジュールで不必要なデータまで宣言する必要がないことを意味しています．その意味では，Cの場合は共通結合というよりは，この後述べる外部結合と考えた方がよいでしょう．ただ，基本的には，Cでの外部変数はどのモジュール(関数)でも使用可能ですので，不本意な使用のされ方をする危険性

は存在します．すなわち，情報隠匿の概念は実現できません．Cの上位言語であるC++では情報隠匿の概念を容易に実現できます．そのためには，クラスを使用することになります．

```
class xyz
  {
        int x;
        int y;
        int z;
    public:
      void modA( );
      void modB( );
  }
```

この例は，xyzという名前のクラスを定義しています．classステートメントからpublicまでの間に宣言されたものはプライベートなものであり，このクラスのなかでしか使用できません．ここでは，データx, y, zがこれに該当します．

x, y, zはこのクラスのなかで宣言されているモジュール(関数)modA，modBでしか使えません．

(3) に関しても，図5.8のような例を考えていただければ，指摘した点が納得してもらえるものと思います．図5.8はC言語で書いたプログラムの一部ですが，この例でwhileは何回繰り返すのかを調べようとしたとき，すぐ理解できるでしょうか．繰り返し条件を指示している変数countが外部変数として宣言されていれば，おそらく，関数func1, func2，func3のどれかがcountの値を更新しているのでしょう．しかし，それはfunc1, func2，func3を全部調べなければわかりません．このように，データを共通域に貯えて使用することは，モジュール間のインターフェース上にそのデータがあらわれないので，処理のしかたが判然とせず，プログラムが大変読みにくくなってしまいます．これは，エラー発生の確率を高め，ソフトウェアの信頼性をそこねる原因になります．

図5.8 何回繰り返すかこれではわからない

```
while (count < = 10) {
        func1 (x, y) ;
        func2 (y, z) ;
        func3 (z, x) ;
}
```

▶共通結合のメリット

このような問題を有しているにもかかわらず，共通結合は，実際にはよく使用されます．その理由は，問題解決のための重要データは，問題構造のどの部分にも関係してくるので，それらを必要のつど，モジュール・インターフェースとしてやりとりするよりも，共通域に貯えておき，どのモジュールからも自由に参照できるようにしたほうが思考の集中がしやすくなるからです．

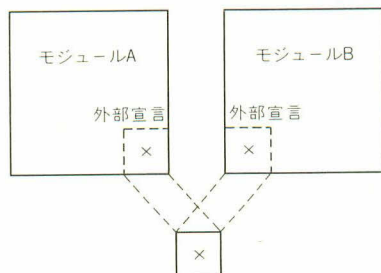
実際，プログラム作成時に関数の引数の数が多ければ多いほど，コーディング時にわずらわしくなる経験は誰でも一度や二度はもっているものと思います．

このように，共通結合は良い面と良くない面の両方をかねそなえています．共通結合を採用するときは，それらの点を十分検討することが大切です．

〔3〕外部結合になるモジュール

「外部結合」は，共通結合とよく似ています．外部結合は，外部宣言をしたデータを複数のモジュールで共有するかたちの結合です．データを共有するという意味では，共通結合とよく似ていますが，共通結合が共通域にあるデータなら，使用してもしなくても，すべて共有したのに対し，外部結合は必要なデータだけを外部宣言するのが普通なので，不必要なデータを共有するといったところがありません．その分だけ，共通結合より結合度が弱くなります．図5.9は外部結合の例を示しています．モジュールA，Bは同じデータXを外部宣言することで共有しています．もちろん，A，BはこのデータXを使用して必要な処理を行うことになります．

図5.6とちがって，使用しないデータまで宣言していない点に注目して下さい．

図5.9　外部結合の概念

外部結合は，共通結合とよく似ていますので，メリット，デメリットもほぼ同じですが，不必要なデータをさらさないので，共通結合の2番目のデメリットはなくなります．

〔4〕制御結合になるモジュール

2つのモジュールの間で，一方が他方に対して引数として制御情報をわたすとき，この2つのモジュールは「制御結合」しているといいます．

制御情報とは，制御スイッチとか機能コードとかいったものであり，わたす側がわたされる側に対して，何らかの制御を指示します．

これは，一方が他方の論理について何かを知っていることを意味し，結合度はそれだけ強いということがいえます．結合度が弱いのは相手のことを何も知らなくても自分の仕事ができるばあいです．すなわち，相手をブラック・ボックスとして扱えるときです．これを可能にするのはデータ結合であり．そのかたちについては後の部分で説明することにします．

▶制御結合の例

制御結合の例としては，論理的強度の部分で説明したテーブルの操作モジュールなどが

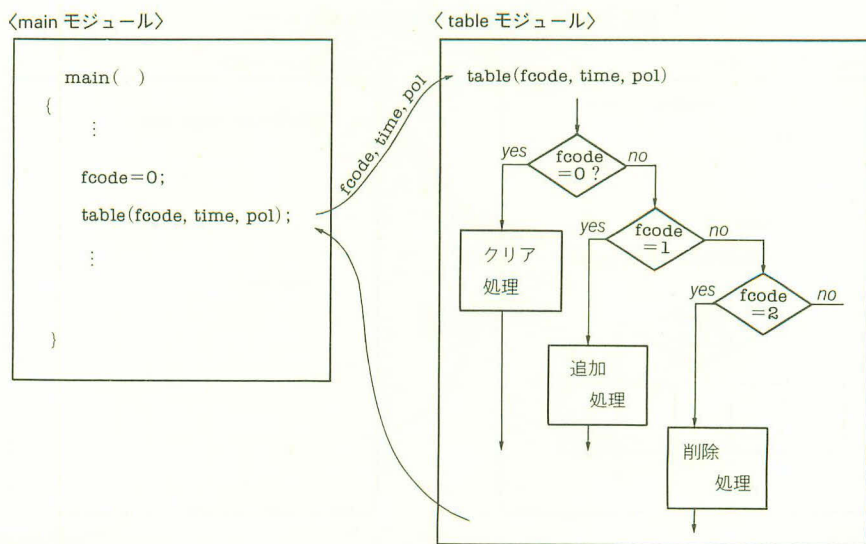
図5.10　論理的強度モジュールの呼出しは制御結合となる

〈main モジュール〉

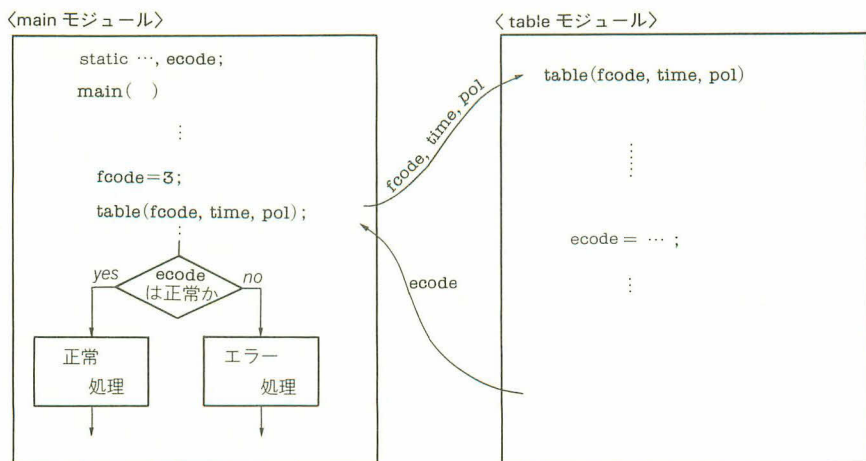
```
main( )
{
  ⋮
  fcode=0;
  table(fcode, time, pol);
  ⋮
}
```

fcode, time, pol

〈table モジュール〉

table(fcode, time, pol)
fcode =0 ?
yes
no
クリア処理
fcode =1
追加処理
fcode =2
削除処理

該当します．テーブルに対して，どのような操作を行うのか，呼出し側(mainモジュール)は，tableモジュールに対し機能コード(fcode)をパラメータで送ることによって指示します．

この例では，呼出し側がtableモジュールの操作内容を知っていて，操作のやり方を指示しています．このような場合，結合度がそれだけ強くなると同時に，呼び出された側(tableモジュール)の強度は論理的強度になってしまい，その意味でもあまり好ましくないのです(図5.10)．また，tableモジュールは，指示された操作の結果，エラー・コード(ecode)を呼出し側に返します．これも制御情報の一種であり，エラー・コードの値によって呼出し側モジュールの処理内容が異なってきます．たとえば，エラー・コードの値を調べて，それが正常であればその操作が正しく行われたものとして，次の正常処理へと進むでしょう．一方，エラー・コードの値がエラーを示すものであれば，エラー・メッセージを書出し，オペレータに次の操作の指示をあおぐといったことになるでしょう(図5.11)．これは，下位モジュールが上位モジュールの処理を指示していることになり，階層構造の基本的特性の一つをおかしていることになります．すなわち，階層構造化の一つの利点は，そのレベル単位で思考を集中でき，他のレベルのことはその時点では考えなくてもよいというところにあります．エラー・コードの例は，この利点をそこなう結果をもたらします．

図5.11 制御結合は階層構造の利点をそこなう

5.2 結合度が弱くなるモジュール化について考えてみよう

前節で述べた4つの結合度の型は，すべて結合度が強くなってしまう例で，その分いろいろな問題点が発生することをみてきました．これらの結合度は，なるべくなら避けたい型です．

それでは，望ましい結合度にするにはどうすればよいでしょうか．望ましい結合度としては，データ結合とスタンプ結合があります．

〔1〕データ結合になるモジュール

「データ結合」は，結合度としてはもっとも弱く，設計結果としては一番望ましいかたちであり，できるだけこのかたちにもっていくように設計しなければなりません．

データ結合とは，スカラ型のデータを引数として2つのモジュール間で，受渡しする型の結合です(図5.13)．

「スカラ型データ」という表現は，異種のデータの集まりとしてのデータ構造との対比で用いられており，1つのデータまたは同種のデータの集まりを意味しています．

図5.12 データ結合は理想的なインターフェース

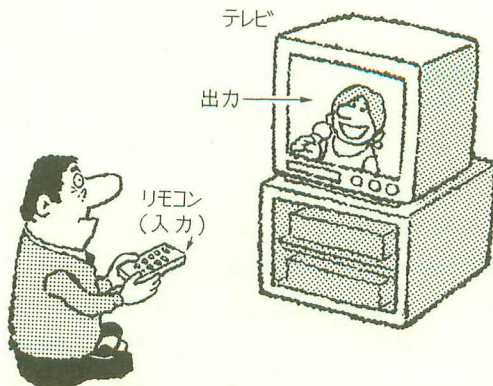



図5.13　データ結合の例

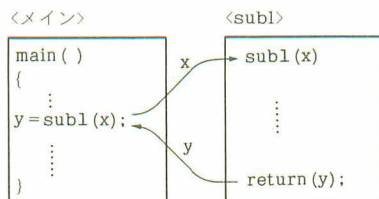



図5.13のように，主モジュールは，入力としてxをモジュールsublにわたし，出力としてyをもらうことさえわかっていればそれで仕事ができ，sublがxをどのようにyに変換するのかという論理を主モジュールが知る必要はありません．このことは，sublの論理を変更しても主側はその影響をうけないということであり，お互いの独立性が高まります．

データ結合では，2つのモジュールが引数を介して直接連絡しあっているので，共通結合や外部結合がかかえていた他のモジュールの変更による影響を最小化することができます．また，使用のしやすさから，再利用の可能性も高まります．

データ結合に対する否定的意見としては，データのうけわたしは，すべてパラメータのかたちで行うためにその数が多くなりすぎ，わずらわしさが増加するというのがあります．しかし，引数がやたらに多くなるのは，構造化そのものに問題があるのが普通です．

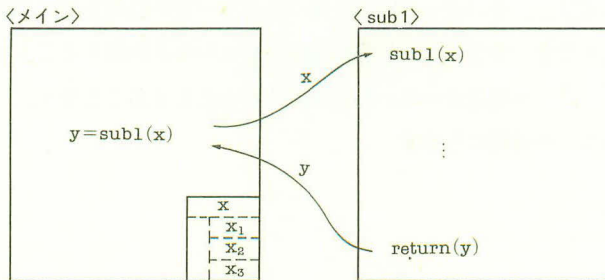
また，パラメータの指定によってモジュール間をいきかう情報が明確になり，それによって理解度が向上するといった良い面を軽視するべきではありません．

〔2〕スタンプ結合になるモジュール

「スタンプ結合」は，2つのモジュール間で必要なデータのやりとりを直接引数で指示する点では，データ結合と同じです．したがって，データ結合がもっている利点はスタンプ結合ももつことになります．

スタンプ結合がデータ結合と異なる点は，パラメータとして指定するデータがスカラ型だけでなく，データ構造も含むことです(図5.14)．複数の異種データの集まりである構造を一つの変数名で表現し，それを引数のなかに含めます．それによって，引数の数を減らせる利点がありますが，共通結合にみられた本来不必要なデータまでうけわたしてしまう(データ構造内の一部のデータは使わない可能性がある)ので，それによるわずらわしさが

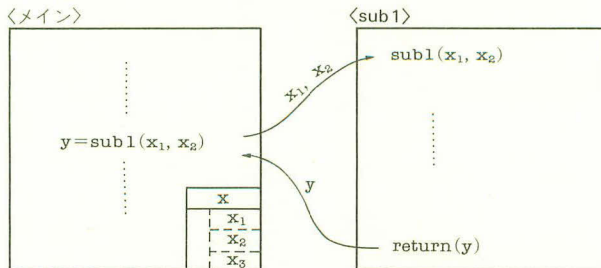
図5.14 スタンプ結合の例

〈メイン〉
y＝sub1(x)
x
x_1
x_2
x_3
〈sub1〉
sub1(x)
return(y)
x
y

生じます．

データ構造を用いたデータベースの使用に関心が高まっていますので，スタンプの使用に関心が高まっています．そして，結合度としては必ずしも悪くはありません．しかし，なるべくなら避けたほうが信頼性の面からは好ましいといえます．

▶スタンプ結合を避ける方法

スタンプ結合になるのを避ける方法は，データ構造のなかで使用するデータだけをスカラ変数として指定する，すなわち，データ結合のかたちに変えてしまうことです．図5.15はその例を示しています．データ構造 x を引数でわたすのでなく，sub1 で実際に使用する x_1，x_2だけをパラメータでわたしています．これは，10 種類程度のデータ集合からな

図5.15 スタンプ結合をデータ結合にする方法

るデータ構造で，使用するのは2，3種のデータだけといったケースに有効です．

もう1つ別の方法は，そのデータ構造をあつかうすべての処理を1つのモジュールにまとめてしまうことです．すなわち，情報的強度モジュールを作成することです．

それによって，データ構造をいちいち引数でうけわたしすることなく，1つのモジュール内で処理することを可能にします．

6．構造化設計に挑戦してみよう

前章までで，構造化設計に対する基礎知識は一応理解できたことになります．簡単に要約すると，

●構造化設計のねらいは，わかりやすいプログラムを作ることにある．

●わかりやすくすることで，作成中のエラーの発生度合を低め，変更や追加も容易にする

●わかりやすさを実現するには，分割，独立性，階層構造化を行う必要がある．

●プログラムを分割したとき，分割した単位はモジュールであり，モジュールには正確な定義と制御ルールが存在する

●モジュールの独立性をできるだけ高めるには，機能的強度とデータ結合の形にするのがよい

●機能をトップ・ダウンで詳細化することで，機能の階層構造ができ，基本的には，それをモジュール構造に対応させればよい．

といったことでした．

さて，これだけの基礎知識を頼りに，とにかく，構造化設計に挑戦してみることにしましょう．

そのために，簡単な例題をとりあげてみます．もちろん，実際の問題はもっと大きくて，複雑ですが，ここでは，問題そのものを正しく理解することが目的でなく，構造化設計の方法をまず理解することが目的ですので，問題は現実のものよりずっと簡略化してあります．

問題を簡略化することで，問題そのものよりも，構造化設計の方に考えの焦点を絞ることができます．それによって，構造化設計のなんたるかがよりよく理解していただけるはずです．

6.1 「入力レコードの妥当性チェック」プログラム

病院の入院患者の状態を監視するプログラムについて考えます．入院患者の脈拍，体温，血圧といった諸要因を随時測定し，データベースに記憶します．また，測定した値が異常(たとえば，体温が38度以上)であれば，看護婦に知らせます．

ここでは，とりあえず測定した値を一定のレコード様式にして入力し，入力レコードとしての妥当性をチェックした後，データベースに記憶する部分に限定して考えてみることにします．

≪プログラム仕様≫

入院患者の測定された諸要因を入力レコードとして読取る．読取ったレコードは，その妥当性をチェックし，正しければ，データベースに記憶する．入力レコードに何らかのエラーがあれば，エラー・メッセージを打出す(図6.1(a))．

この病院には，AとBの2種類の病陳があり，Aには重症患者，Bには軽症患者が入院している．入力レコードは，病陳ごとに別個に設計されており，その詳細仕様は図6.1(b)に示されている．

入力レコードの妥当性チェックは，つぎのような項目に対して行うものとする．

レコード・コード：‘T’

レコード・タイプ：‘A’または‘B’

測定日：

DD　31以下

MM　12以下

YY　91または92

測定時間：

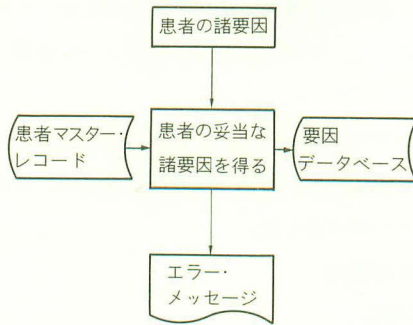
TT　24以下

MM　60以下

患者番号：患者マスタとの比較

測定諸要因：数字データ

担当医師，看護婦コード：英数字データ

図6.1　入力レコードの妥当性チェック・プログラムの仕様

（a）概略図

● 患者要因レコードA（A病棟）

コード	タイプ	測定日	測定時間	患者番号	脈拍	体温	最高血圧	最低血圧	皮膚抵抗	担当医師コード	看護婦コード
T	A	DDMMYY	TTMM	XXXX	XXX	XX.X	XXX	XXX	XXX	XXXX	XXXX
(1)	(1)	(6)	(4)	(4)	(3)	(3)	(3)	(3)	(3)	(4)	(4)

● 患者要因レコードB（B病棟）

コード	タイプ	測定日	未使用	患者番号	脈　拍	体　温	最高血圧	最低血圧	皮膚抵抗	未使用	未使用
T	A	DDMMYY	−	XXXX	XXX	XX.X	XXX	XXX	XXX	−	−
(1)	(1)	(6)	(4)	(4)	(3)	(3)	(3)	(3)	(3)	(4)	(4)

● 患者マスタ・レコード

コード	患者番号	患者名	住所	電話番号	病棟	担当医師コード	看護婦コード	ベッド番号
T	XXXX	XXX---XX	XX---XX	XX—XX	AまたはB	XXXX	XXXX	XXXX
(1)	(4)	(10)	(20)	(10)	(1)	(4)	(4)	(4)

（b）レコード仕様

さて，この仕様から，プログラムで行うべき機能について整理してみましょう．

プログラム全体としては，患者の諸要因レコードを正しく読取り，データベースに記憶することです．そこで全体機能をつぎのようにまとめることにします．

「患者の妥当な諸要因を得る」

このプログラムの構造化設計を行うさいに，どこから手をつけていけばよいでしょうか．

いままで学んだ知識を活用すれば，構造化設計の第一歩は，プログラムを分割し，いくつかのモジュールを作成し，それを階層構造化することでした．そのさい，モジュールの独立性をできるだけ高くすること，そのためには，モジュールを単一機能としてまとめることでした．

このような方針で，プログラムをモジュールに分割するためには，先程のプログラムの全体機能をトップ・ダウンで展開し，機能の階層構造を作り，それをモジュール構造に対応させればよいこともすでに学びました．

そこで，プログラムの全体機能

「患者の妥当な諸要因を得る」

はどのように機能展開できるかを考えてみましょう．まず，問題仕様をよく読むことです．それによってこの全体機能を行なうためには必要な機能の概要が把握できます．思いつくままに，おもなものをあげればつぎのようになります．

- 患者のマスタ・レコードを読取る
- 患者の諸要因を読取る
- レコード様式の妥当性をチェックする
- エラー・メッセージを打出す
- レコードをデータベースに記憶する

こまかなところまで考えれば，まだいくつかの機能があげられるかもしれませんが，大きな機能はまあこんなところが常識的でしょう．

これらのことから，機能階層図をまとめると図6.2のようになるでしょう．このような図に整理することで，プログラムで行おうとしていることが，文章で説明されているとき

図6.2 プログラム仕様の機能展開

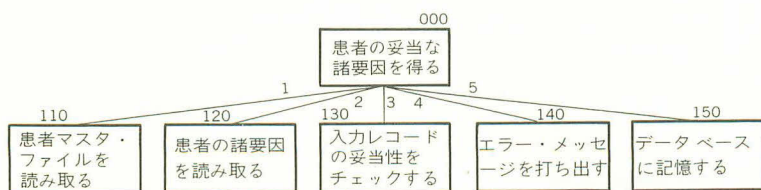



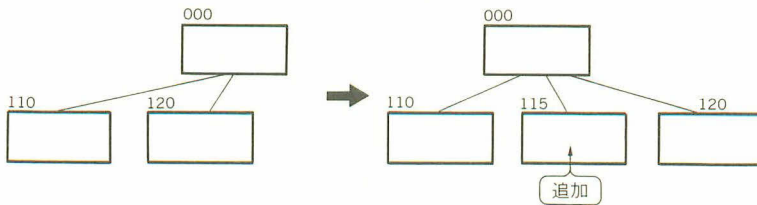
図6.3 モジュール番号と追加処理

よりも，ずっと理解しやすくなったと思いませんか．ひとつの図は千の文章より価値があるというものです．

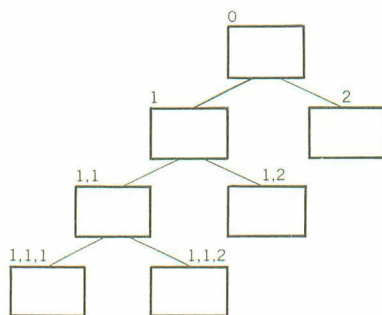
図6.2では，各機能の長方形枠の上辺に3桁の番号が付けてあります．これは機能番号(モジュール構造図ではモジュール番号)です．この番号を用いることで各機能を多くの機能のなかから識別するのが容易になります．また，番号の付け方を体系的に行うような工夫をすることで，各機能の階層構造上での位置づけを明確にすることができます．この例で用いた3桁の番号は，頭1桁が階層構造のレベルを表わしています．また，下1桁が0にしてありますが，こうしておくと後で追加があったとき，そこに有効数字を用いることで，全体の体系をくずさずに追加することができます．

たとえば，レベル1の110と120の機能の間に1つの機能を追加したいときは，番号として115を使えば110，115，120と続き異和感がありません(図6.3)．

もちろん，番号の付け方はこの方式だけでなく，他の方式もあります．たとえば，図6.4のように，階層構造のレベルごとに番号の桁を増やしていけば，機能間の従属関係が大変明確になります．その他，読者の皆さんもいろいろ工夫して使ってみて下さい．

少し話が横道にそれましたが，もとへ戻しましょう．図6.2の機能展開ができたので，これらの機能間のインターフェース・データを定義してみます．モジュールの独立性を高くするには，機能的強度にするほかに，データ結合にすることが必要でしたから．

図6.2で，まず，インターフェース1について考えてみます．「患者のマスター・ファイルを読取る」は，患者マスター・ファイルを読取り，読取ったファイルを親機能である患者の妥当な諸要因を得る」にわたす機能を行います．したがって，インターフェース・

図6.4　体系的な番号のつけかた

データは「患者マスター・ファイル」になります．インターフェース・データは下位機能を基準に考えますので，この場合は出力データということになります．インターフェースの入力データ(上位から下位へわたすデータ)は特にありません．

インターフェース2についても，1と同様に考えることができます．「患者の諸要因を読取る」は，患者要因レコードを読取って，親機能にわたします．したがって，インターフェース・データの出力として「患者要因レコード」が定義できます．入力データは，この場合もありません．

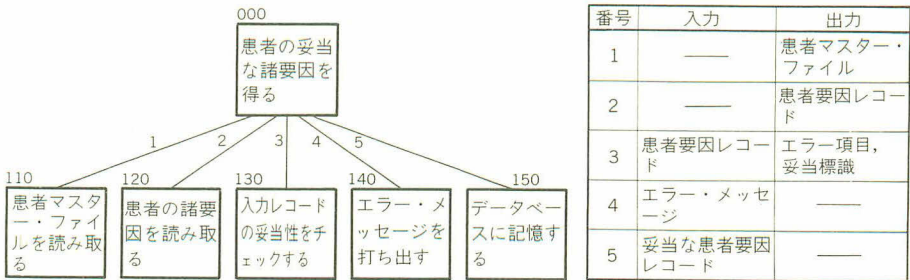
インターフェース3はどうでしょうか．「入力レコードの妥当性をチェックする」機能は，親機能からチェックするべき患者要因レコードをもらいます．これはインターフェース2で親機能がうけとったものです．受取った要因レコードの妥当標識(エラーがなければ0，エラーがあれば1)を親機能に戻します．したがって，インターフェース・データとしては，入力が「患者要因レコード」，出力が「エラー項目」と「妥当標識」になります．

インターフェース4，5に関しては，同じような考え方をしていけば，もう自分で定義できますね．練習のために少し自分で考えてみて下さい．図6.5に構造図とそのインターフェース・データの定義をまとめて示します．インターフェース・データを定義することで，このプログラムで行おうとしていることが，より明確に理解できたと思います．

さて，機能展開を続けましょう．問題仕様を詳しく読むと，このプログラムの重要な部分は，入力レコードのチェックにあることがわかります．

そこで，図6.5の「入力レコードの妥当性をチェックする」機能(130)をさらに詳細に展開してみることにします．まず，妥当性をチェックするべき入力レコードとして，2つのタイプがあることが問題仕様に述べられています．A病棟の患者用のレコードとB病

図6.5 インターフェース定義

番号	入力	出力
1	——	患者マスター・ファイル
2	——	患者要因レコード
3	患者要因レコード	エラー項目，妥当標識
4	エラー・メッセージ	——
5	妥当な患者要因レコード	——

棟の患者用のレコードの2種類です．病棟ごとにレコードが分けられているのは，A病棟は重症患者用であり，そのために要因の測定もひんぱんに行われなければならず，また，測定データの値によっては，患者の状態が危険であり，すぐに担当医師や看護婦にそのことを知らせる必要がでてきます．したがって，A病棟レコードには，B病棟には特に必要としない測定時間，担当医師，看護婦といったデータが特につけ加えられているのです．

これらのことから，入力レコードの妥当性をチェックする機能は，2つの入力レコード・タイプごとに妥当性をチェックすることとして，図6.6のように展開することができます．この部分は，問題のキーになっている重要なところなので，もう少し詳しく分析してみます．問題仕様にも，妥当性のチェックの仕方の詳細が述べられています．特に，ここで重要なのは，やはり患者番号の妥当性のチェックです．患者を監視することがこのプログラムの目的なのですから，まず，入力レコードの患者番号が正しいかどうかをチェックすることが第一です．

図6.6 「入力レコードの妥当性をチェックする」の機能展開

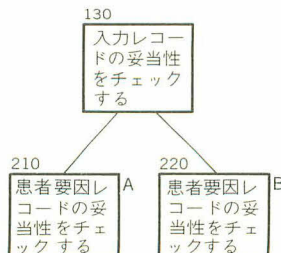
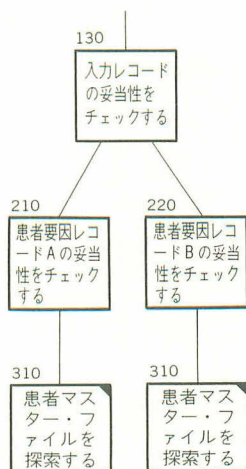



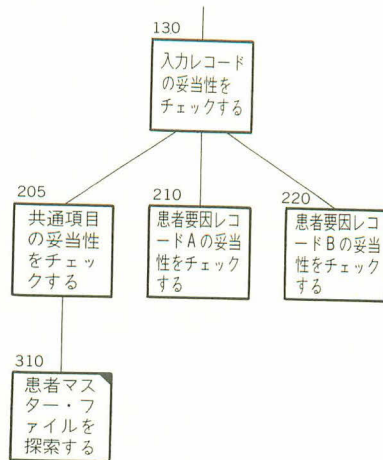
図6.7　図6.6の拡張

このチェックの方法として，患者マスター・レコードにあらかじめ登録してある患者番号と入力された患者番号とを比較する方式を採っています．そのため，患者マスター・ファイルを読みとる必要があります．このファイルの読取りは，図6.5で作成した構造図では，別機能として前もって実行することになっています．そこで，「入力レコードの妥当性をチェックする」機能では，すでに読取ってあるマスター・ファイルのなかに，入力された患者番号が存在するかどうかを確かめるためにファイルを探索することになります．

この探索機能は，入力レコードがAタイプでもBタイプでも必要です．そこで，この機能は共通機能として独立させ，必要に応じて呼出すようにします．結果は図6.7になります．これは図6.6の機能構造図を拡張したものです．

入力レコードの妥当性チェックの内容をもう少し続けましょう．患者番号の他に，レコード・コード，レコード・タイプ，測定日，測定時間，測定諸要因，担当医師，看護婦コードといった項目がチェック対象になっています．これらのチェック項目は，Aタイプと Bタイプとを比較してみれば，両タイプに共通したものと片方のタイプだけのものが混在しています．

両タイプに共通した項目は，その妥当性のチェック方法も両タイプで同じですから，それを各タイプごとで行うと当然重複します．そこで，この部分を独立させ1回だけ実行させればよいように工夫します．図6.8に工夫した構造図を示します．

この構造図と図6.7の構造図とを比較してみて下さい．図6.7では，上に述べたように，共通項目の妥当性のチェックは210と220の両方で重複して行われることになります．コ

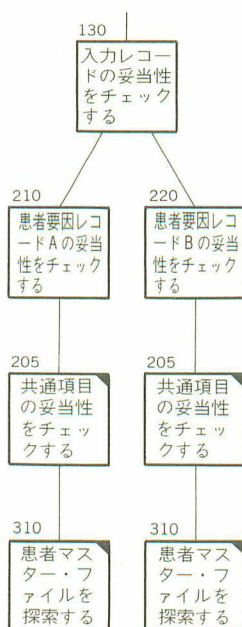
図6.8 図6.7の修正案

ーディングすれば，210と220に同じパターンのルーチンが存在することになるでしょう．これは無駄な話です．しかし，図6.8では，それを避けることができます．「入力レコードの妥当性をチェック」(130)するときは，入力レコードがA，Bどちらのタイプであろうと，まず，「共通項のチェック」(205)を行い，その後，タイプに応じて，210，220でタイプ固有の項目のチェックを行ないます．

この方法によって，「共通項目のチェック」はコーディング時に1回行うだけでよいことになります．この図で，「患者マスター・ファイルを探索する」(310)も従属機能として定義されていることに注意して下さい．その理由はわかりますね．そうです．患者番号のチェックも両タイプの入力レコードに共通するものだからです

このように，図6.8の構造図は図6.7の構造図が有していた問題点を解消します．その意味では，すぐれた構造図です．しかし，この図は何かおかしく感じませんか．この構造図にしたがってプログラムを作成した場合，おそらく，正常に稼動するはずです．それでは，何がおかしいのでしょうか．

それは，この構造図を機能展開図としてみたときのおかしさです．先に述べたように，機能を階層構造に展開したとき，上位機能はそれに従属する下位機能の総括になっていなければなりません．

しかし，図6.8の構造図はそのようになっていません．たとえば，「患者要因レコードAの妥当性をチェックする」は，その文章が意味するごとく，レコードAのすべての項目の妥当性をチェックしてはじめてそう言えるのです．図6.8では，Aタイプに固有の項

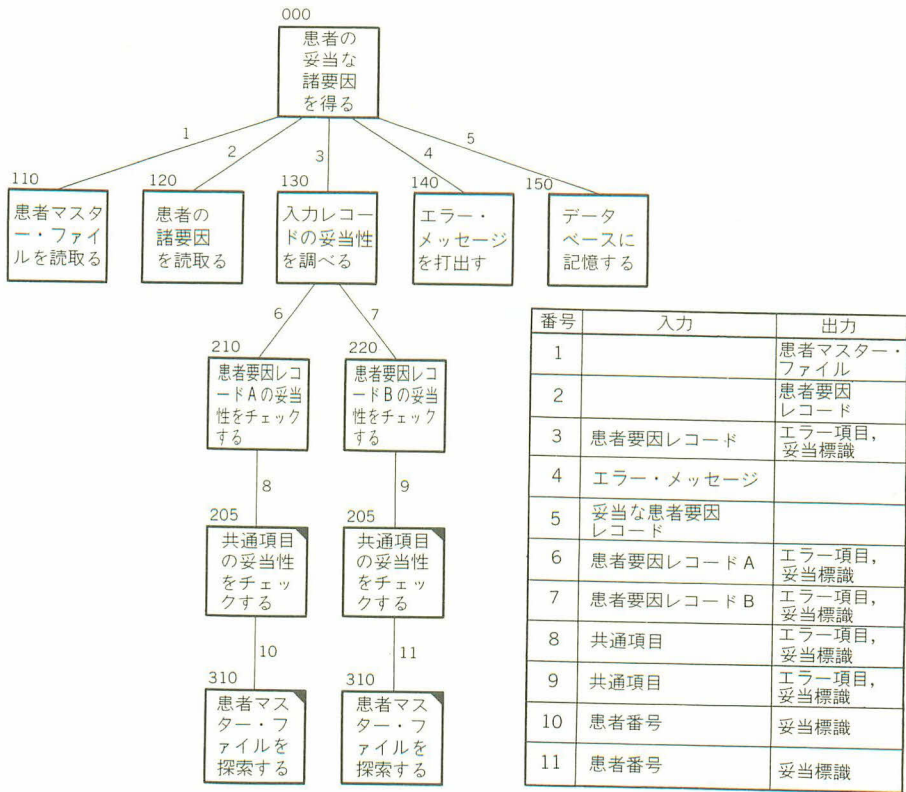
図6.9　図6.8の問題点を解消

目だけをチェックし，共通項目は同レベルの別にある隣接機能で行います．これでは，看板にいつわりありで，誤解をまねきやすく，ミスの原因になります．

ここは，もう少し素直な構造図に修正する必要があります．代替案として，図6.9のような構造図はどうでしょうか．「共通項目の妥当性をチェックする」を共通機能にし，それをレコードAとBの妥当性をチェックする機能のそれぞれ(210と220)に従属機能として定義しています．こうすれば，先の問題点は解消しますね．図6.10にこの問題の全体構造図とインターフェース・データを示します．

6.2　構造化設計の手順を正しく理解しよう

前節では，いままで身につけた構造化設計の知識を頼りに，提示された問題に対して，とりあえず挑戦してみました．そして，図6.10のような結果を得たわけですが，この経験をふまえて，構造化設計の手順を整理し，正しく理解することにしましょう．

図6.10 「患者の妥当な諸要因を得る」の機能階層図とインターフェース最終案

番号	入力	出力
1		患者マスター・ファイル
2		患者要因レコード
3	患者要因レコード	エラー項目，妥当標識
4	エラー・メッセージ	
5	妥当な患者要因レコード	
6	患者要因レコードA	エラー項目，妥当標識
7	患者要因レコードB	エラー項目，妥当標識
8	共通項目	エラー項目，妥当標識
9	共通項目	エラー項目，妥当標識
10	患者番号	妥当標識
11	患者番号	妥当標識

6.2.1 基本的な手順をまず知っておこう

▶モジュール定義と機能分割

図6.11は，構造化設計の手順の概要です．設計の手順の最初は，プログラムの最上位モジュールを定義することからはじまります．最上位モジュールの定義は，通常，そのプログラムが全体として行おうとしている機能そのものの定義になります．たとえば，大気汚染度を測定するプログラムの設計では，「大気汚染度を測定する」になります．また，先程の問題では，「患者の妥当な諸要因を得る」が最上位モジュールになります．

最上位モジュールの定義ができたら，つぎはこのモジュールの機能を解くべき問題としてながめ，どのような従属機能に分割できるのかを考えていくことになります．

分割していくときには，ある機能を親と子といった縦の関係，すなわち階層構造の上下

図6.11　構造化設計の手順

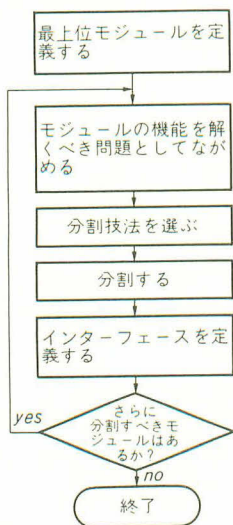
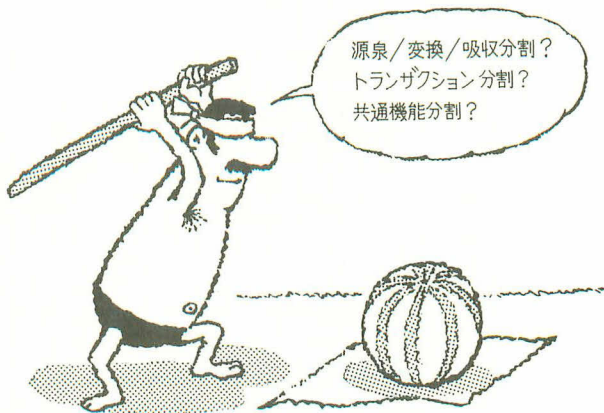
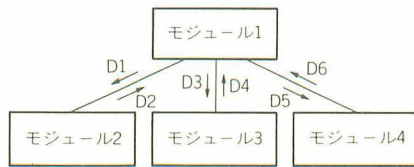



図6.12　機能分割の技法はいろいろある

関係に分割することと，兄弟，すなわち階層構造の同位レベルのいくつかのものに分割することの両方を考えていかなければなりません．もちろん，親子にするか，兄弟にするかは，問題の内容というか，機能の内容に依存します．これは前節の問題で，そのいくつかの例を検討しましたね．そのときには特にふれませんでしたが，機能分割にさいして使用できるいくつかの技法があります．源泉/変換/吸収分割，トランザクション分割，共通機能分割などと呼ばれている技法です．これらの技法については，この後順を追って説明していくことにします(図6.12)．

これらのうちから，適切な分割技法を選択し適用することにより，直接従属する1組のモジュールが明らかになってきます．

▶インターフェースの定義

モジュール間の直接従属関係が定義できれば，つぎにそれらモジュール間のインターフェースを定義します(図6.13)．インターフェースを定義するということは，モジュール間に受渡しされる入力と出力の情報の種類を明らかにすることです(図6.14)．このことも，前節ですでに経験しましたね．

図6.13 モジュール・インターフェースの定義

このような過程を階層構造の上位から下位へと順を追って繰り返していきます．もちろん，この繰返し過程は，どこかで終了しなければなりません．もし，この分割過程をやりすぎると，最後は1命令しか含まないモジュールになってしまうでしょう．機能を入力から出力に変換する処理とみるかぎり，1命令でも十分機能とみなせるからです．

しかし，モジュールを一つの命令にまで分割するのは，明らかにやりすぎで現実的ではありません．

それ以上の分割をやるべきか，やめるべきかの判断は，そのモジュールをコーディングしたときにどのくらいの命令数で書けるかを考えてみることです．一つの基準として，命令数が50程度の大きさであると考えられるときは，そのモジュールは，それ以上分割する必要がないと思ってさしつかえないでしょう．なぜなら，普通の人なら50命令程度の論理ならとくに苦労せず頭に思い描けるでしょうし，思い描ける以上は，それで十分わかりやすい設計が実現できたということになるからです(**コラムC参照**)．

6.2.2 源泉/変換/吸収分割

解くべき問題をモジュールに分割するときのもっとも基本的な技法が源泉/変換/吸収分割です．

源泉(Source)とは，問題を解くための源となるもの，すなわち入力のことです．**変換**(Transform)は，入力を変換して出力に変えることを意味しています．**吸収**(Sink)とは，問題の結果を吸収すること，すなわち出力のことです．

プログラムを設計するときの基本は，そのプログラムで解くべき問題の構造とプログラムそのものの構造とを一致させることです．

この分割技法は，その意味で問題固有の構造を発見し，それをモジュール構造を作りあげるときの基本にしようとする考えにたっています．

問題構造の発見のためには，データの流れに注目しデータが問題構造のなかをどう流れるかを理解することが大切です．データが問題構造のなかを動くにつれてどう変換されていくか，すなわち入力から出力へ変換される過程をつぶさに分析することで問題構造を明らかにし，それに対応したモジュール構造を作りあげます．結果として，モジュールは機能的強度をもつことになります．

この分割技法の手順は図6.14のようになります．

前節の問題を検討する段階では，この分割手順はまだ知らなかったので，かなり我流の機能分割をやってしまったわけですが，この手順にそって行うとどうなるのか，もう一度おさらいのつもりで考えてみましょう．それがまた，この手順をしっかり理解するためにも役立つはずです．

図6.14の第1ステップは，問題構造の大要を部分機能で表現することです．解くべき問題を，たとえば3～10個くらいの部分機能で表現してみます．部分機能をとりあげるときは，問題をデータの流れにそって分析し，主要なものをとりあげるようにします．

このことは，実は前節で無意識のうちに行っていましたね．問題の全体機能「患者の妥当な諸要因を得る」を行うために，主要な部分機能として，次の5つをあげました．

コラムE

モジュールの大きさ

モジュールの大きさはどのくらいが適当なのか，本文中では50命令(行)ぐらいとしました．その理由としては二つあります．

第一は人間が頭のなかで識別できるパターンは，個人差がありますが$2^{5\pm2}$くらいだといわれています．50というのはその意味でごく平均的な人が頭のなかでロジックを描ける平均的な大きさだというわけです．皆さん自身の経験ではどうでしょうか．

もう一つの理由は，プログラムのソース・リストを印刷するとき，印刷用紙の1ページ内におさまる命令が約50だということです．プログラムを検討するとき，1ページ内にすべてがおさまっていれば考えやすいですね．

もっとも，50説には反論もあります．1,000命令くらいのプログラムをモジュールに分割するときは，モジュールの数は20個ですみますが，10,000命令になるとモジュールの数は200個になり，個々のモジュールの管理が大変になります．1個1個のモジュールの役割を理解するのも大変ですね．そのような場合は，実際には100～200命令程度のモジュールが作られることがよくあります．

図6.14 源泉/変換/吸収/分割の手順

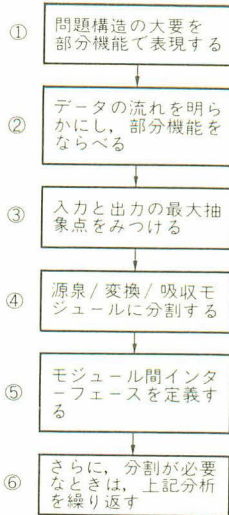
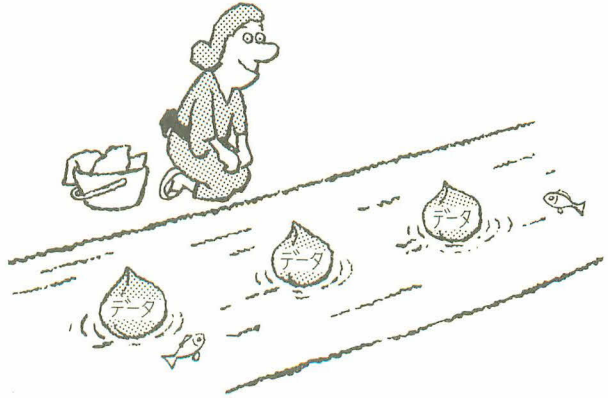



図6.15 データの流れに注目

(1) 患者マスター・レコードを読取る

(2) 患者の諸要因を読取る

(3) 入力レコードの妥当性をチェックする

(4) エラー・メッセージを打出す

(5) 入力レコードをデータベースに記憶する

このうち，(1)，(2)は入力機能，(3)は主要処理機能，(4)，(5)は出力機能であり，データの流れにそってあげてあることがわかります．これはたまたまそうなったのではなく，人間が問題解決のために，その問題を分析するとき，入力→処理→出力の過程で考えるのはごく自然だからなのです．

このことをもう少し整然と整理しようとするのが，この手順の第2ステップになります．

第2ステップでは，問題のなかの主要な入力と出力データの流れを明確にし(図6.15)，それにそって，第1ステップでとりあげた部分機能をならべます．すなわち，問題に対するDFDを作成します．

たとえば，先の問題では，図6.16のようになるでしょう．このようにならべれば，問題のなかでは，データの流れが1つだけではないことがわかります．しかし，この手順では，そのなかの主要な入力と出力のデータの流れに注目します．その観点からみれば，こ

の問題での主要なデータの流れは，患者の諸要因を読取り，その妥当性をチェックし，妥当であればそれをデータベースに記憶するところにあります．

患者マスタ・ファイルを読取ったり，エラー・メッセージを打出したりするのは，あくまでも，この問題の主要なテーマを解決するときに付随的に出てくる入出力と解釈するべきでしょう．

▶最大抽象入力点と最大抽象出力点

分析の第3のステップは，問題構造のなかで入力と出力に対する**最大抽象点**を見つけることです．

最大抽象点とは，入力または出力の抽象化が最大になる点のことであり，ここでいう“抽象化”とは，最初はっきりしていた入力がいくつかの処理をほどこされて，もはや入力とみなせなくなること，または，はじめて出力らしき形が現れることを意味しています．前者を**最大抽象入力点**，後者を**最大抽象出力点**と呼びます．

これら各点を見つけ出す理由は，問題をもっとも独立性の高い複数の部分機能に分割させるためです．

最大抽象入力点を見つけるには，第2のステップでならべた問題構造の出発点からはじめて，入力の流れを見つけながら問題構造のなかに入っていくと，入力データがどんどん抽象化されて，もはやこれ以上は入力とみなせなくなる点に達するはずです．そこが最大抽象入力点になります．

一方，最大抽象出力点を見つけるには逆に問題構造の終点から出発し，出力の流れを見つめながら問題構造のなかをさかのぼっていくと，出力はだんだん抽象化された形になってきて，ついには出力の流れが最初に現れる点に到達します．ここが最大抽象出力点です．

このことを例題にあてはめてみましょう．最大抽象入力点は，「患者の諸要因を読取る」と「入力レコードの妥当性をチェックする」の間になります．その理由は，前者の機能の実行結果として得られるのは入力データであり，後者の機能は出力を求めるための処理であるからです．

一方，最大抽象出力点は，「入力レコードの妥当性をチェックする」と「データベースに入力レコードを記憶する」の間になります．後者はあきらかに出力処理であり，その出力は前者機能が行われた後にはじめて姿をあらわします(図1.17)．

これら2つの最大抽象点を見つけ出せば，問題を3つのもっとも独立した機能，すなわち源泉(入力)，変換(入力から出力へ)，吸収(出力)機能に分割できることになります．問

図6.16 部分機能をデータの流れにそってならべる

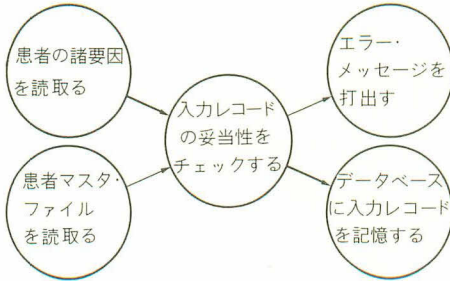
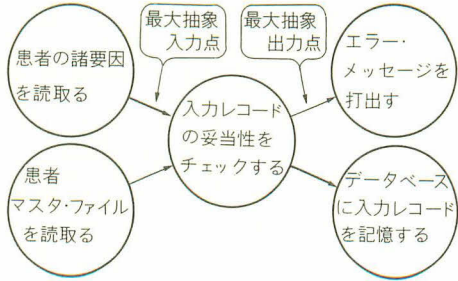



図6.17 最大抽象点をみつける

題構造の開始点から最大抽象入力点までが源泉機能であり，一般的に表現すれば，入力データの流れを最大に抽象化したかたちで得る機能ということになります．

つぎに，最大抽象入力点と出力点にはさまれた部分が変換機能であり，最大に抽象化された入力データを出力データの最大に抽象化されたかたちに変換する機能ということができます．

3番目は，最大抽象出力点と問題構造の終点までの部分が吸収機能であり，出力データを要求されている形で出力装置に出力する機能ということになります．

これら3つの機能は，それぞれ単一な機能としてあらわされ，これらの機能をもった3つのモジュールを最上位モジュール(問題全体の機能)の直接従属モジュールとして定義します(図6.18)．そして，最上位モジュールと直接従属モジュール間のインターフェースも定義します．

このことを例題に適用すれば，図6.19のような結果になります．この結果は，先に検討した構造図と多少異なっています．たとえば，図6.5の結果と比較してみて下さい．図6.5の結果は，源泉/変換/吸収分割を特に意識していないので，部分機能として同一レベ

図6.18 源泉/変換/吸収モジュールへの分割

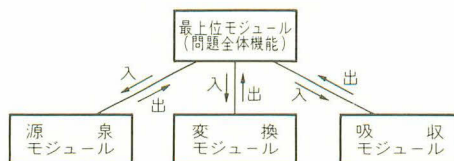
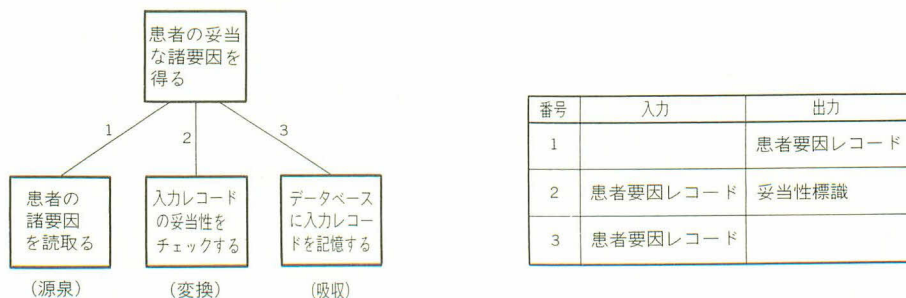



図6.19 患者の妥当な諸要因を得る」の源泉/変換/吸収モジュールへの展開

番号	入力	出力
1		患者要因レコード
2	患者要因レコード	妥当性標識
3	患者要因レコード	

ルに5つの機能をもつています．これはこれで決して間違っているわけではないのです．しかし，図6.19の結果と較べると，やや問題の焦点がぼけている点はいなめません．先に検討した情報隠匿の概念からいけば，図6.5はこの段階としてはやや詳細情報をさらけ出しすぎているようです．

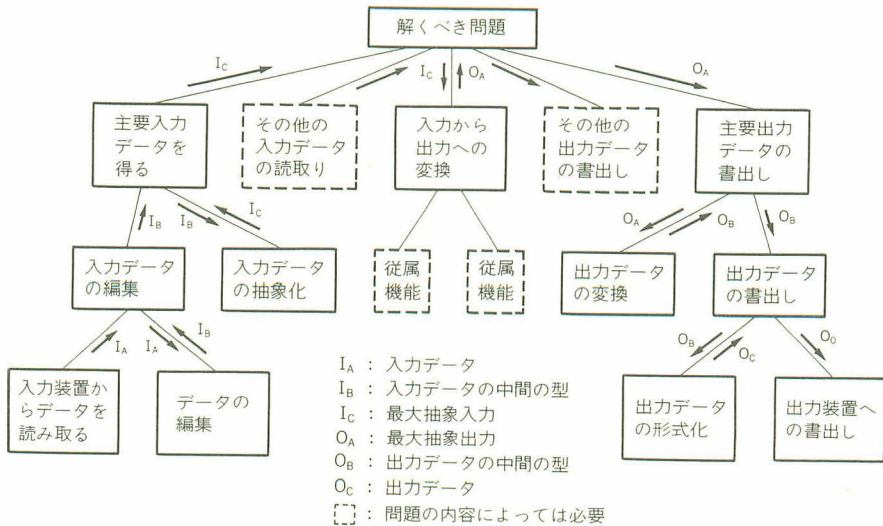
その点，図6.19の結果は，問題の主要な入力，出力とその処理に焦点を絞っているので，この問題の主機能として何を行うべきかが簡潔に表示されています．

ただ，この例題のように，多くの問題では，主要入出力の他に別のデータの流れをもっています．例題では，「患者マスタ・ファイルを読取る」，「エラー・メッセージを打出す」がこれにあたります．これらの処理はさけるわけにはいきませんので，その処理のために，結果として，図6.5のような構造図になってしまうことはよくあります．しかし，マスター・ファイルの読取りやエラー・メッセージの打出しを必ずこのレベルで行う必要があるかというと，それは別問題です．

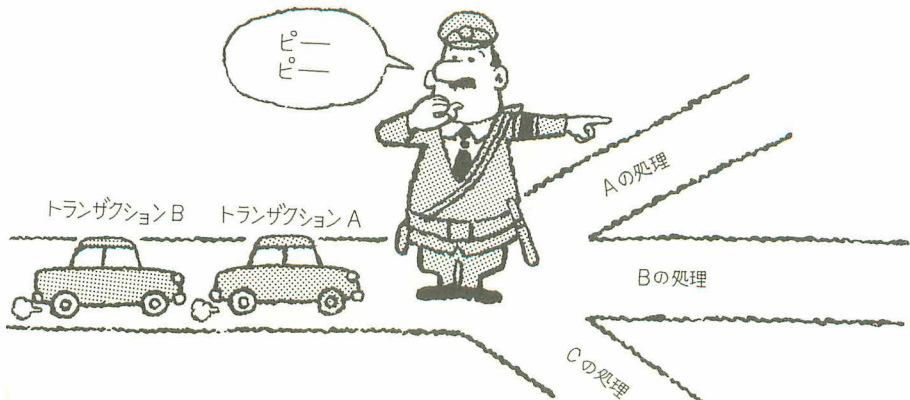
特に，この例題では，「エラー・メッセージの打出し」は，「入力レコードの妥当性をチェックする」の下位機能として定義した方がよさそうです．理由は，先の情報隠匿の面からとインターフェース・データの簡潔さの面からです．この場合は，上位機能に対して，エラー項目をわたさずにすみます．

問題の全体機能を源泉，変換，吸収機能へ展開できれば，後は，それらをさらに分割する必要があるかどうかを検討します．分割の必要があると認められれば，分割を続けます．

もっとも，すべてのケースに源泉/変換/吸収分割技法が適用できるとは限りません．適用できるのは，問題の型が，入力→処理→出力になっている場合です．
もちろん，大部分の問題はこの型に属しますので，この分割技法はそれだけ汎用性をもっ

図6.20 入力/変換/出力問題の一般的構造

ています．この型の一般構造は図6.20のようになるので参考にして下さい．

問題によっては，このような分割技法では解決できないものもあります．その場合は，他の分割技法を用いることになります．

図6.21 トランザクション分割

6.2.3　トランザクション分割

トランザクション分割とは，入力トランザクションにいつくかの種類があり，その種類に応じて別個のモジュールを設けるやり方です(図6.21)．源泉/変換/吸収分割が問題構造を入力，変換，出力に応じてモジュール化したのに対し，トランザクション分割は，変換機能のモジュールをさらに細かく分割するときによく用います．

たとえば，3種類のトランザクションa，b，cを処理するプログラムでは，モジュール分割は，一般に図6.22のようになるはずです．この図の階層構造の上位部分の分割は，源泉/変換/吸収分割技法によって，下位部分はトランザクション分割技法によって分割されています．もちろん，この種の問題の前提として，トランザクションの種類ごとにちがった仕事を実行するものと考えています．

トランザクション分割は，モジュールの独立性を高めます．なぜなら，たとえば，トランザクションaを処理するモジュールは，トランザクションaの処理について注意を払う唯一のモジュールであり，他の種類のトランザクションb，cについては，いっさい考える必要がないからです．b，cを処理するモジュールについても同様のことがいえます．

この分割の考え方は，実は，先の例題で経験しましたね．例題では，患者要因レコードA，患者要因レコードBがトランザクションでした．そして，入力レコードの妥当性をチェックする機能で，トランザクションごとにその妥当性を検討するように設計しました．この部分がトランザションン分割です(図6.23)．こうすることによって，互いに相手を意識することなく，それぞれのタイプごとに処理ができることになります．また，それぞれのトランザクションに何らかの変更があったとき，そのトランザクションの処理モジュールだけを変更すればよく，他のモジュールへの波及効果を防ぐことができます．

図6.22　変換機能に対するトランザクション分割

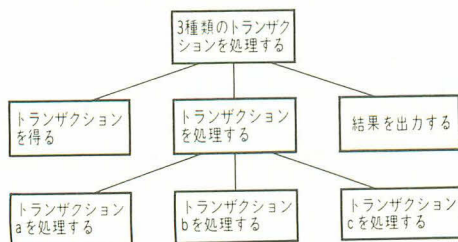
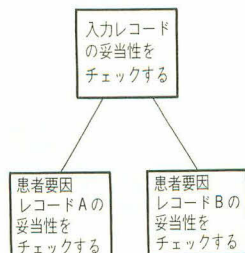



図6.23　例題におけるトランザクション分割

6.2.4　共通機能分割

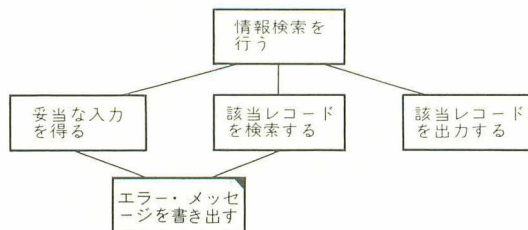
共通機能分割は，解くべき問題を源泉/変換/吸収分割やトランザクション分割などを用いて分割していく過程で，定義したいくつかのモジュールに共通の機能を見いだせるとき，その機能をとり出し別個のモジュールとして定義するやり方です．そして，その共通機能を必要とするモジュールは，必要なときに別個に定義したモジュールを呼び出すようにします．

図6.24は，その１つの例です．入力を読み取るときに何らかのエラーがあれば，エラー・メッセージを打出してエラーがあったことを知らせます．また，入力が要求したデータがデータ・ファイルになければ，その旨をエラー・メッセージとして伝えます．「エラー・メッサージを書出す」機能は，両者共通の機能ですのでこれを共通機能モジュールとして別個に定義し，必要に応じて「該当入力を得る」モジュール，または「該当レコードを得る」モジュールで呼び出します．

共通機能モジュールを定義することで１個のプログラム内でおなじ機能を重複させることを避けることができます．これはトランザクション分割と同様に，変更時にその波及効果を局所化できるという利点にもつながります．

共通機能には，もう１つのとらえ方があります．それは，１つのデータ構造やテーブルに関連する機能は，そのデータ構造やテーブルに関する共通機能であると考え，一つのモジュール内ですべて処理するようにします．これは情報隠匿を可能にし，やはり，変更の波及効果を局所化できる利点があります．この考えで定義したモジュールは，強度的には情報的強度をもつことになります．

第４章で論じた大気汚染度テーブル(表4.2)に関するすべての機能を１つのモジュールで処理するようにした例(図4.4)は共通機能分割になっています．

図6.24　共通機能分割の例

6.3　患者監視プログラム

構造化設計の正しい手順が理解できたことと思います．そこで，再び元に戻って，本章の冒頭でとりあげた患者監視プログラムを全体のまとまった問題として構造化設計してみることにしましょう．

6.1では，種々の分割の仕方をみるために，この問題の入力部分をかなり脚色して検討しました．実際には，患者の諸要因は，患者が寝ているベッドにそなえつけたアナログ装置によって直接読取ることになるでしょう(図6.25)．

6.3.1　問題の仕様

《患者監視プログラム》

病院の患者を監視するプログラムを設計する．各患者はアナログ装置によって監視される．このアナログ装置によって，脈拍，体温，血圧，皮膚抵抗といった要因を測定する．プログラムは(各患者に対して指定された)一定時間間隔でこれらの要因を読み取り，データベースに記憶する．各患者の各要因に対する安全範囲は，前もって規定されている(たとえば，患者Xの正常な体温は36.6～37.5℃の範囲というように)．ある要因が患者の安全範囲をはずれたとき，あるいはアナログ装置が正常に作動しなかったときは看護婦控室に通知される．なお，患者ごとの監視時間間隔のリスト，患者番号とベッド番号の対応，患者ごとの安全範囲データの3つは，あらかじめファイルとして準備されている．

図6.25　患者監視システム

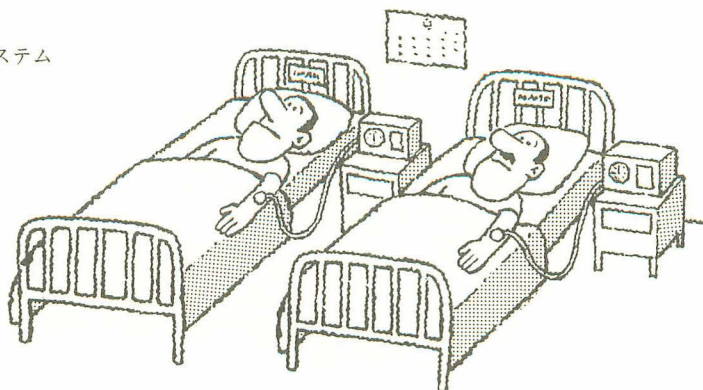
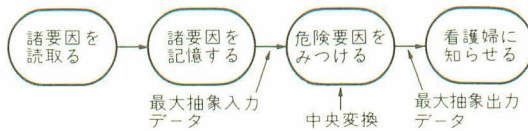



図6.26 「患者監視プログラム」の問題構造

6.3.2 問題の分析

すでに理解したように，複合設計の最初のステップは問題構造の概略を把握するために，問題仕様から部分機能をいくつか取り出してみることです．

そして，それらを主要な入出力データの流れにそってならべてみることです．この問題の主要な入力は，アナログ装置から読み取られる患者の諸要因(体温，脈拍など)でしょうし，出力は患者に異常が発見されたときの看護婦への警報です．したがって，とりあげた部分機能をこれらのデータの流れにそってならべた結果は，図6.26のようになります．

▶最大抽象点とモジュール定義

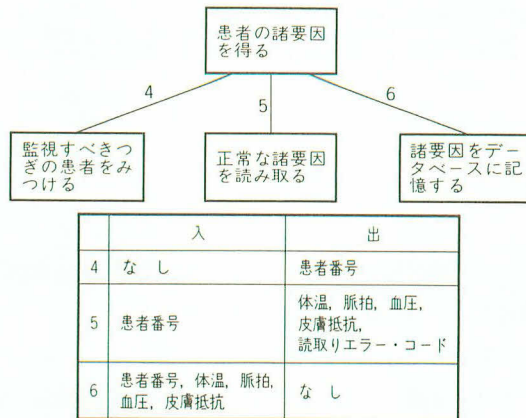
また，最大抽象点は，入力に関しては読み取られた患者の諸要因が安全範囲データと比較できるような形式になる点です．出力に関しては，患者に対する危険要因(安全範囲外のデータが発見された要因)が出力になるわけですから，それらが最初の形であらわれるところです．図6.26にこれらが同時に示されています．

ここまで分析がすすめば，モジュール構造の設計を始めることができます．まず，トップのモジュールの機能は，解くべき問題そのものを機能的に表現したものですから，「患者の監視を行う」になります．そしてそれらに直接従属するモジュールとしては，源泉モジュールとして「患者の諸要因を得る」，変換モジュールとしては「危険要因をみつける」，吸収モジュールとして「危険要因を看護婦控室に知らせる」が定義できます(図6.28)．

これらのモジュール間のインターフェースもあわせて図6.27に示してあります．源泉モジュールでは一般に親モジュールからもらう入力データはありません．出力データはもちろん読み取った患者の諸要因です．

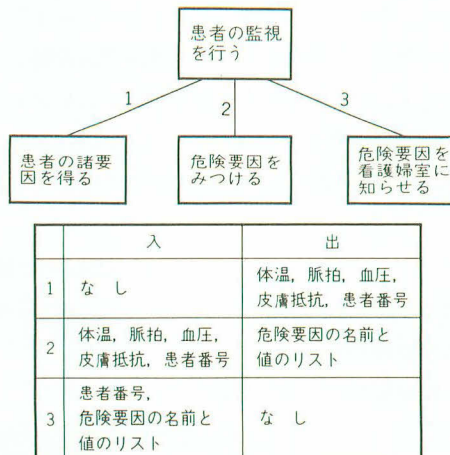
変換モジュールでは，源泉モジュールから親モジュールへわたされた患者の諸要因を逆に親モジュールからもらうことになります．そして危険要因がみつかったときはそれを親モジュールにわたします．入力を出力に変換するために，変換モジュールは読み取られた

図6.27　モジュールのトップ構造

	入	出
4	な　し	患者番号
5	患者番号	体温，脈拍，血圧，皮膚抵抗，読取りエラー・コード
6	患者番号，体温，脈拍，血圧，皮膚抵抗	な　し

諸要因とあらかじめデータベースに貯えておいた安全範囲データとを比較することになります．

吸収モジュールでは，患者番号とその危険要因を親モジュールから入力としてもらい，それを看護婦控室の端末装置に出力します．吸収モジュールから親モジュールへわたす出力は一般にはありません．

図6.28 「患者の諸要因を得る」の分割

	入	出
1	な　し	体温，脈拍，血圧，皮膚抵抗，患者番号
2	体温，脈拍，血圧，皮膚抵抗，患者番号	危険要因の名前と値のリスト
3	患者番号，危険要因の名前と値のリスト	な　し

▶源泉モジュールの分割

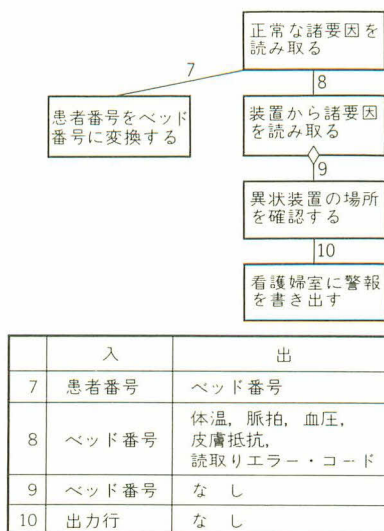
つぎに，源泉モジュール「患者の諸要因を得る」をもう少し細かく分割してみましょう．問題仕様からこの機能は3つの部分機能からなりたっていることがわかります．

- つぎに監視する患者をきめる(患者ごとに指定された一定時間間隔にもとづく)．
- アナログ装置から諸要因を読み取る．
- 読み取った諸要因をデータベースに記憶する．

したがって，図6.28のようなモジュール展開を行うことができます．図6.29にはモジュール間インターフェースも定義してあります．入出力データの個々についてはとくに説明するまでもなく理解してもらえると思いますが，インターフェース番号5の出力，「読取りエラー・コード」についてだけ注釈しておきましょう．これは，アナログ装置が何らかの理由で正常に作動せず諸要因が読み取られなかったときに，それを親モジュールに知らせるために設定されたものです．

このことをさらに正しく理解するために，「正常な諸要因を読み取る」機能をもっと詳しく分割してみる必要があります．このモジュールでは，監視すべき患者の番号を入力としてもらいます．そしてそれをベッド番号に変換することが必要になります．というのは

図6.29 「正常な諸要因を読み取る」の分割

	入	出
7	患者番号	ベッド番号
8	ベッド番号	体温，脈拍，血圧，皮膚抵抗，読取りエラー・コード
9	ベッド番号	な　し
10	出力行	な　し

病院に固定しているのはベッドであり，患者はそのときどきで変わります．したがって，アナログ装置はベッドに固有のものであり，患者に固有のものではないのです．

ベッド番号がきまれば，そのアナログ装置から諸要因を読み取ることになります．もしそのときにアナログ装置が正常に作動しないときは，そのベッドの設置場所を看護婦控室に知らせます．この結果を図6.29に示します．図6.29，「装置から諸要因を読み取る」モジュールの下辺の菱形記号は，その下の異状装置の場所を確認するモジュールが条件(エラーがあったとき)に応じて実行されることを示しています．

▶変換モジュールの分割

変換モジュール「危険要因をみつける」では，先にものべたように，行うべきことは個々の患者に対する安全範囲をデータベースから読み取ることと，患者から読み取った諸要因をそれらと比較し，安全範囲外のものがあるかどうかを調べることです．したがって，モジュール分割を行うと図6.30のようになります．

▶吸収モジュールの分割

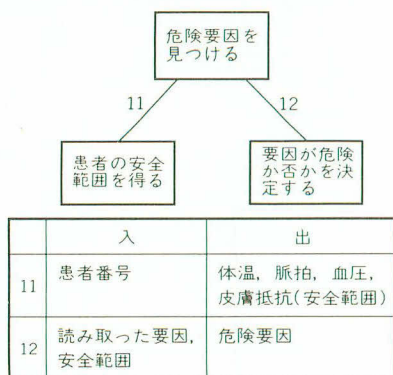
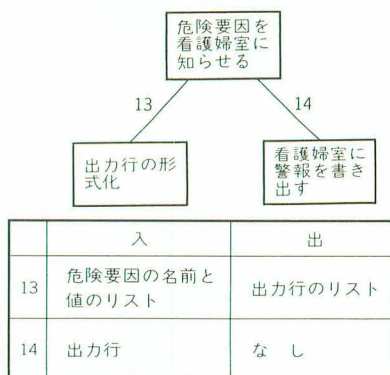
吸収モジュール「危険要因を看護婦室に知らせる」についての分割は，とくに説明することもないと思います．結果だけを図6.31に示しておきます．

図6.30 「危険要因を見つける」の分割

	入	出
11	患者番号	体温，脈拍，血圧，皮膚抵抗(安全範囲)
12	読み取った要因，安全範囲	危険要因

図6.31 「危険要因を看護婦に知らせる」の分割

危険要因を看護婦室に知らせる
13
14
出力行の形式化
看護婦室に警報を書き出す

	入	出
13	危険要因の名前と値のリスト	出力行のリスト
14	出力行	な　し

6.3.3　結果に対する評価

以上でこの問題に対するモジュール分割作業は，一応完了したことになります．この設計結果に対して１つの問題点があります．それは先にふれた読取りエラー・コードに関するものです．この読取りエラー・コードはインターフェース５と８の出力データとしてあらわれていますが，これは一種の制御情報です．

「装置から諸要因を読み取る」モジュールは，装置が正常に作動しなかったときのコード値を設定し，親モジュール「正常な諸要因を読み取る」にわたします．正常な諸要因を読み取るはさらにインターフェース５を介して親モジュール「患者の諸要因を得る」モジュールにこのコードをわたします．「患者の諸要因を得る」は，このコードの値の結果によって「監視すべきつぎの患者をみつける」モジュールを呼ぶ(エラーのとき)か，「諸要因をデータベースに記憶する」モジュールを呼ぶ(正常のとき)かを決定します．

これらの事実は，これに関連する３つのモジュールが，データ結合でなく制御結合になっていることを示しています．制御結合は先にのべた理由によりあまり好ましい型ではありません．そこで，先に得られたモジュール構造の修正が必要になります．読取りエラー・コードをインターフェース・データにあらわれないようにするにはどうすればよいでしょうか．それは読取りエラーに関連する処理を複数モジュールで行うのではなく，特定のモジュール内で処理できるように分割方法を変えればよいのです．

修正した結果を図6.32に示します．装置からの読取りを「つぎの患者の諸要因を読み

図6.32　構造の修正

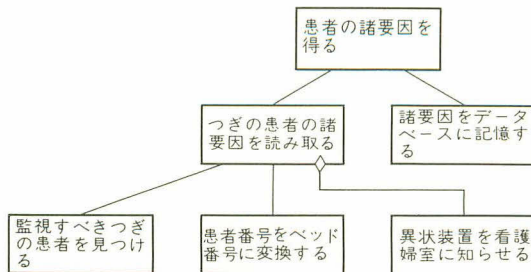
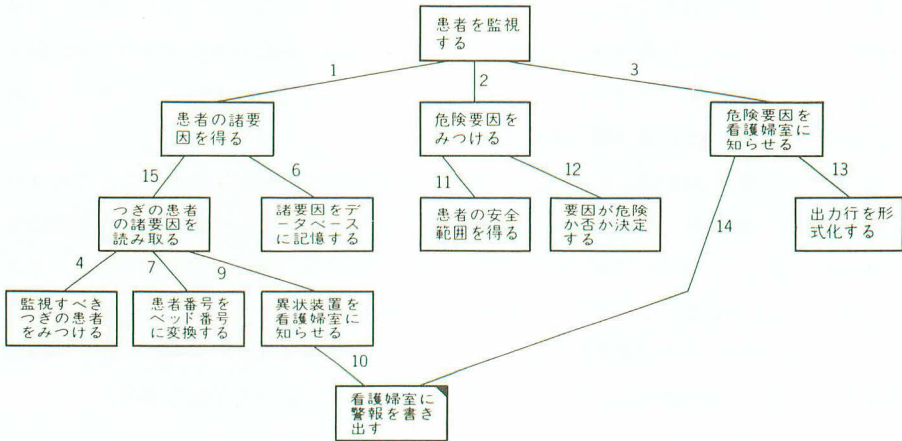



図6.33 「患者監視プロログラム」の全体構造

	入	出
1	な し	体温, 脈拍, 血圧, 皮膚抵抗, 患者番号
2	体温, 脈拍, 血圧, 皮膚抵抗, 患者番号	危険要因の名前と値のリスト
3	患者番号, 危険要因の名前と値のリスト	な し
4	な し	患者番号
6	患者番号, 体温, 脈拍, 血圧, 皮膚抵抗	な し
7	患者番号	ベッド番号
9	ベッド番号	な し
10, 14	出力行	な し
11	患者番号	(体温, 脈拍, 血圧, 皮膚抵抗)の安全範囲
12	読み取った諸要因, 安全範囲	危険要因
13	危険要因の名前と値のリスト	出力行のリスト
15	な し	体温, 脈拍, 血圧, 皮膚抵抗, 患者番号

取る」で行うようにすることが問題解決のキーです．こうすることによって「監視すべきつぎの患者を見つける」モジュールを呼ぶか，親モジュール「患者の諸要因を得る」にコントロールを戻すかは自己モジュール内で判断でき，読取りエラー・コードをインターフェース・データとして他のモジュールにわたす必要がなくなります．修正後の設計結果の全体を図6.33に示します．これらの結果が機能的強度とデータ結合の形になっていることをよく確かめてください．

6.4　設計結果に対する評価

この章でみてきたように，構造化設計の基本は問題を機能的に構造化し，それをモジュール構造に対応させていくところにあります．

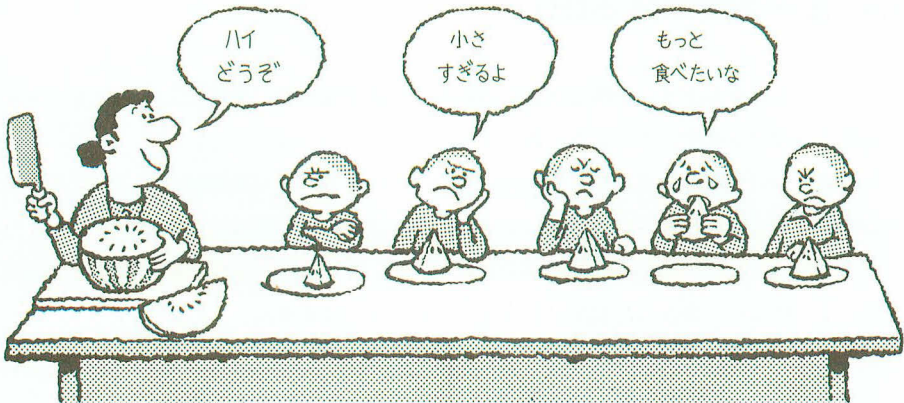
それは1つの方法論とみることができますが，だからといって料理読本のようにサジ一杯の味加減まで規定しているわけではありません．設計過程をおおまかに枠組みしたものにすぎません．したがって，同じ問題を設計しても，誰が行っても同じ結果になるとは限りません．むしろ，異なった結果になる方が普通かもしれません．

したがって，設計結果に対してはたしてそれが最良の結果であるのかどうかをいろいろな角度から見てみる必要あります(図6.34)．その場合，つぎのような観点から評価してみることができます．

- モジュールの大きさ
- モジュールの独立性
- 階層構造としての的確さ
- 入出力処理の隔離とインターフェースの簡潔さ
- 制御範囲と影響範囲

以下，これらの点について個々にみてみることにしましょう．

図6.34　設計の結果はいろいろな面から評価する

図6.35　大きさはよく検討して決定しよう

▶モジュールの大きさ

第2章で，わかりやすいプログラムにするための設計の基本方針について述べました．その第1は，分割の概念でした(図6.35)．

分割することで，問題解決のために同時に考えなければならない要因の数を少なくすることができます．その意味では，モジュールの大きさは小さいほうがよいと言うことができます．

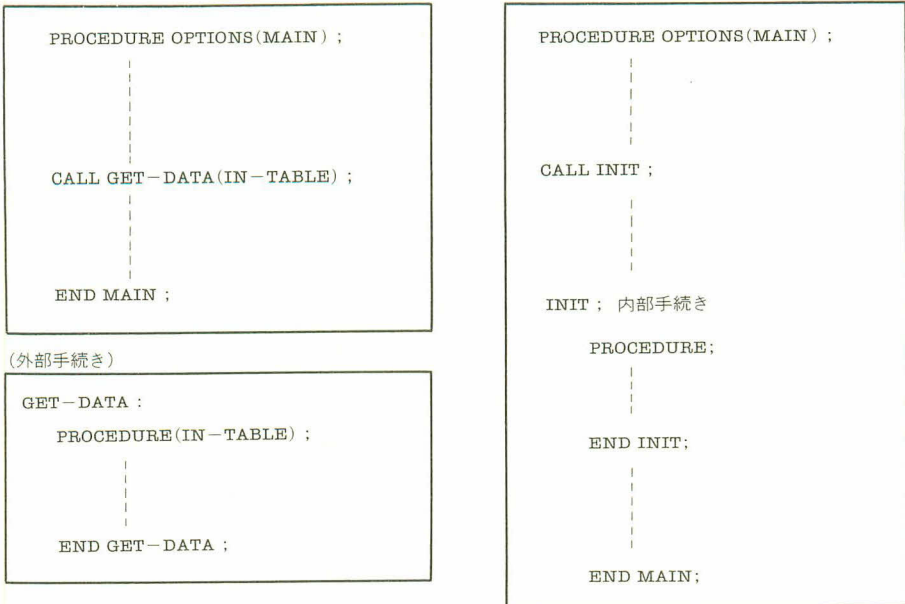
人間が一度に理解できる限界は$2^{5\pm2}$通りくらいという説があります．これをモジュールの大きさに適用すれば，命令数で30くらいということができます．また，プリンタ・リストの1ページ(約50命令)内におさまるくらいの大きさにしておけば，モジュールを調べるとき，いちいちページめくりをしなくてよいから理解しやすいという実現的な考えもあります．

ただ，現実にモジュール化を行なうとき，30とか50という数は，モジュールの大きさとしては，やや小さすぎるきらいがあります．

たとえば，命令数が1000のプログラムで，モジュールの大きさを50命令とすれば，20個のモジュールができます．これが10000命令のプログラムになれば，モジュール数は200個になります．これらのモジュールをすべて独立して扱い，管理するのは大変やっかいな話です．

実際には100〜200命令の大きさのモジュールが多くみられます．

構造化設計の提唱者の一人であるG.J.Myersは，命令数が2000〜3000のプログラムで

図6.36　PL/I 言語でのモジュールとセグメントの違い

(外部手続き)

(a)　外部手続きは単独にコンパイルできる(モジュールの例)

(b)　内部手続きは MAIN モジュールと一緒にコンパイルする(セグメントの例)

は，モジュールの大きさは 40～50 命令に，10000 命令以上のプログラムでは，100～150 命令にするよう勧めています．

比較的小さいプログラム(たとえば，1000 命令以下)では，個々に独立してコンパイルするモジュールよりも，分割した部分機能単位をセグメントとして扱うほうがより実際的と言えます．

セグメントとは，

- 命令が語彙としてもともっている
- 固有の名前がついていて他から呼出し可能である

の 2 つの条件を満足するものです．これはモジュールの条件から

- 単独でコンパイルできる

という条件をとり除いたものと考えることができます．

モジュールとセグメントを区別して考えるのは，プログラム言語によっては，それがまったく違ったものを対象にするからです．

たとえば，PL/I 言語ではモジュールは外部手続きに相当しますが，セグメントは内部手続きに相当します．外部手続きは単独でコンパイルできますが，内部手続きはできません．そして，これら2つの実際的な価値のちがいは，呼び出し時にデータを引数としてうけわたしできるかできないかという点にあります．外部手続きは引数指定ができますが，内部手続きはできません(図6.36)．

このことは，内部手続きを用いるときは，データ結合でなく，共用結合になってしまうことを意味しています．このことは，COBOL 言語のサブプログラムと PERFORM パラグラフに対してもあてはまります．

しかし，本書の読者の方々が多く使用されているC言語では事情が異なってきます．

Cでは，モジュールに該当するものは関数ですし，セグメントに該当するのも関数と考えることができます．関数を単独でコンパイルするか，そのプログラムの一部として，他の部分と一緒にコンパイルするかは，その関数の用途によってきまります．したがって，Cではモジュールでもセグメントでも，データを引数としてうけわたしできることになります．

モジュールの大きさは，テストのしやすさにも大いに関係してきます．簡単な例で考えてみましょう．

図6.37は比較的大きなモジュールAの論理を表わしています．このモジュールのすべての論理経路をテストするためには，12(2×3×2)のテスト・ケースをテストする必要があります．

このモジュールAを，いま，図6.38のように2つのモジュールB，Cに分割したとき，すべての論理経路をテストするためには，Bが6(2×3)ケース，Cが2ケースになり，あわせて8ケースですむことになります．分岐数が多くなるほどこの差は大きくなります．

モジュールの大きさを考える時のいくつかの考慮点について述べてきましたが，基本的には，モジュールは問題の機能構造に起因するものであることを忘れないでいただきたいと思います．モジュールの大きさは，どんな場合でも50命令とか100命令以内に収めなければならないと強制的に考えるのはまちがっています．

先にあげた数字はあくまでもおおまかな目安であると思って下さい．その目安を基準にして，あまりに大きすぎるモジュールは，もう少し部分機能に分割できないかどうかを検討してみて下さい(図6.39)．

また，反対にあまり小さすぎるモジュールは分割しすぎでないか検討してみて下さい．特にそのモジュールを呼び出す親モジュールが1つしかないようなときは，親モジュール

図6.37
すべての経路をテストするには12とおりのテスト・ケースが必要

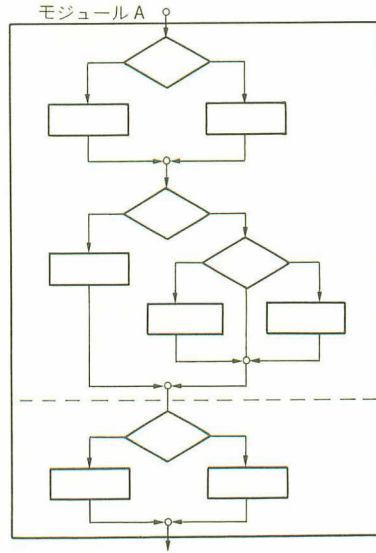



図6.38 あわせて8とおりのテスト・ケースですべての論理経路がテストできる

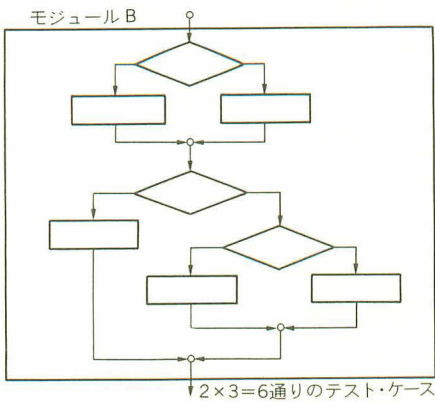
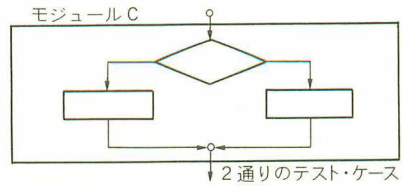



図6.39
大きすぎるモジュールは部分機能を別モジュールに

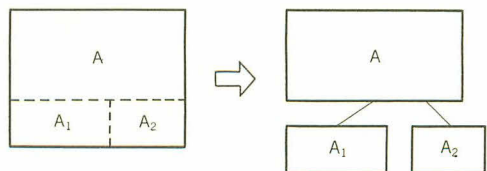
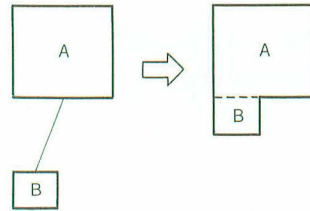



図6.40　小さすぎるモジュール(B)の統合

との統合を検討してみて下さい(図6.40)．

多くのモジュールから呼び出されているモジュールの場合は，統合すればそのモジュールの部分が呼び出しモジュールに重複して持たれることになり，保守作業であやまりをおかす可能性が高くなりますので気をつけて下さい(図6.41)．このようなときは，小さくても独立したままにしておくことが得策でしょう．

ただ，独立させておくと，それを呼び出す時のオーバヘッド時間が問題になるときがあります．呼び出し数が多ければ多いほど，それは問題になってきます．

そのようなときの解決策は，その呼び出し数の多い小さなモジュールをあくまでも独立させたまま，インラインのモジュールとして，親モジュールに統合させることです(図6.42)．

独立させたモジュールとして扱うことで，そのモジュルに変更があったとき，一箇所の変更だけですませることが可能になります．もし，親モジュールに重複させることで統合してしまえば，重複分だけの変更が必要になります．

▶モジュールの独立性

モジュールの独立性については，４章と５章で詳しく検討しました．結論として，モジ

図6.41　多くのモジュールから呼び出されるモジュールを親モジュールに統合する

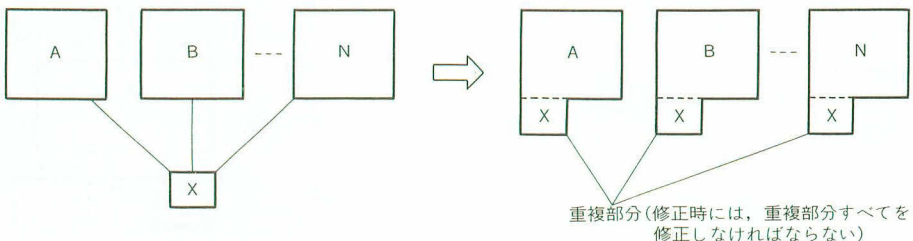
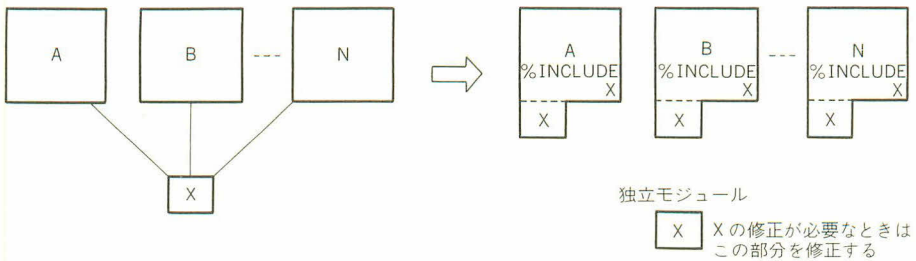
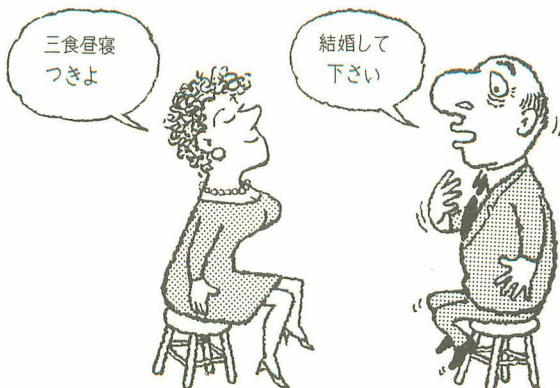



図6.42 親モジュールの多いモジュールを統合するときはインライン扱いにする

ュール強度は機能的強度か情報的強度に，モジュール間結合度はデータ結合にするのが望ましいと述べました．

しかし，どんなときでも必ずそのような結果になるようにしてしまうということではありません．構造化設計の真の目的は，よく考えて設計するということにあります．データ結合と共有結合の長短をよく理解した上で結合度を設計し，よく検討した結果が共有結合の方がよいとなればそれはそれでよいのです．避けなければならないのは，何の方針ももたずに設計することなのです(図6.43)．

ところで，定義したモジュールの強度や結合度がどのようなタイプになっているかを判断するにはどうしたらよいのでしょうか．

結合度に関しては，第5章の内容をよく理解すれば，判別するのはそれほど難しいこと

図6.43 独立性はよく考えて

ではありません．しかし，強度に関しては，慣れないと判断に苦しむことがよくあります．

強度を判定する簡単な方法はモジュールの定義文をチェックしてみることです．たとえば，G. J. Myers はつぎのように言っています．

- その文が重文で，コンマがあるか，または複数個動詞があれば，そのモジュールはたぶん複数個の機能を実行している．したがって，それは手順的，連絡的，あるいは情報的であることが多い．
- その文に，時間的経過に関係がある言葉，たとえば，「まず」，「つぎに」，「さらに」，「…のとき」などの言葉が含まれている場合，そのモジュールはたぶん手順的である．
- 初期値化，消去などの言葉があれば，時間的である可能性が強い．
- その文の述語に単一の特定の目的語がないばあい，そのモジュールは論理的であることが多い．たとえば，「すべてのデータを編集する」といった表現である．

モジュール強度が機能的であるためには，少なくとも上のような表現がとられていず，文の述部にたいして明確な1つの目的語が存在することです．

このような判定方法の他に，第4章で検討した内容をもとに，つぎのような面から強度の判定を行うことができるでしょう．

・モジュール機能の記述のしにくさ
・モジュールの複数機能の実行
・呼び出されたときの機能の実行数
・機能と入口点との対応
・モジュール内の機能の機能的関連性
・モジュール内の機能の手順的関連性
・モジュール内の機能のデータ的関連性

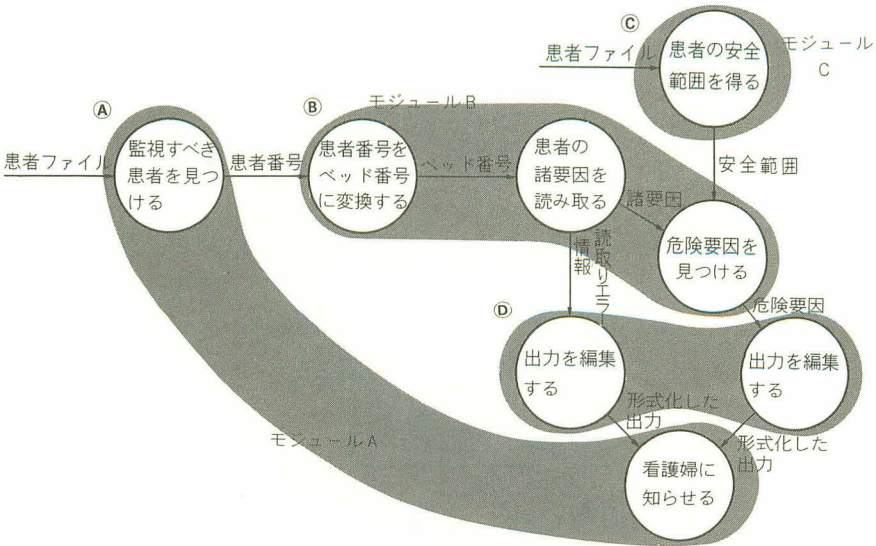
これらの要素とモジュール強度との関係をまとめたものを図6.44に示しておきます．

強度判定の例を簡単な具体例をもとに考えてみましょう．図6.45は前節でとりあげた患者監視プログラムの1つの設計例です．実際には，この図のバブルごとに示された個々の機能単位にモジュール化することになる(すべて機能的強度になる)でしょうが，ここでは

図6.44 モジュール強度の判定

評価項目＼強度	暗合的	論理的	時間的	手順的	連絡的	情報的	機能的
モジュール機能を記述しにくい	Y	N	N	N	N	N	N
モジュールは複数機能を実行する	Y	Y	Y	Y	Y	Y	N
呼び出された時は1個の機能だけ実行する	−	Y	N	N	N	Y	−
各機能ごとに入口点をもっている	−	N	−	−	−	Y	−
機能的に関連した複数機能を実行する	N	Y	−	−	−	Y	−
手順的に関連した複数機能を実行する	−	−	N	Y	Y	−	−
データ的に関連した複数機能を実行する	−	−	−	N	Y	Y	−

Y：該当する
N：該当しない
−：無関係

図6.45 患者監視プログラム

強度判定の練習が目的ですので，図のアミ部分のようなまとめ方で4つのモジュールに分けたとします．

モジュールAは「監視すべき患者を見つける」と「看護婦に知らせる」の2つの機能をまとめたものです．このモジュールを簡潔な言葉では定義できませんね．

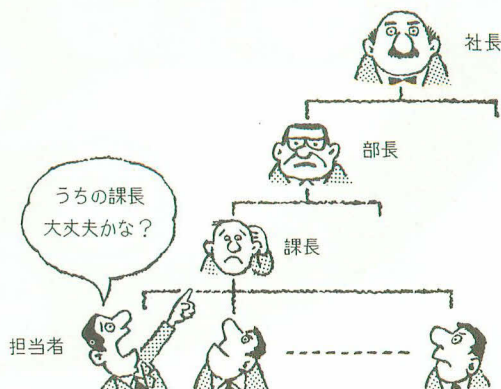
また，2つの機能は機能的にも，手順的にも，データ的にも関連性は認められません．したがって，このモジュールは暗合的強度になります．

モジュールBは3つの機能「患者番号をベッド番号に変換する」，「患者の諸要因を読取る」，「危険要因を見つける」を実行します．これら3つのモジュールは出力が後の機能の入力として使われています．したがって，このモジュールは連絡的強度になります．

モジュールCは1つの機能「患者の安全範囲を得る」を実行します．このモジュールは機能的強度をもちます．

4番目のモジュールDは，「エラー・メッセージを出力編集する」機能と「危険要因を出力編集する」機能をまとめて1つのモジュールにしています．このモジュールの強度はいろいろな見方ができます．1つの見方は共通機能をまとめたものとして機能的強度とみる考え方です．この見方があてはまるのは，エラー・メッセージと危険要因の形式が同じであり，このモジュール内で編集処理がまったく同じであるときです．

入力ごとに編集処理が異なるときは，論理的強度か情報的強度になります．入力ごとに入口点をもっていれば情報的強度，入口点が1つであれば論理的強度になります．

図6.46　階層構造の的確さも検討する

▶階層構造の的確さ

わかりやすいプログラムにするための設計の第3の概念は階層構造化でした．

階層構造化することで，問題を段階的に考え，発展させていくことができます．特に，問題を機能的に階層構造化することは，先に2つの概念，分割と独立性を実現させるのことにもなります．

その意味では，階層構造としての的確さを評価する第1の視点は，問題構造として正しい機能階層構造になっているかどうかです(図6.46)．正しい機能の展開は理解を容易にします．上位機能は下位機能の総括になっているかを注意してチェックする必要があります．機能と手順をまちがって把えていないかは，この機能の上下関係をみることで明らかになります．

図6.47は，この機能の上下関係が正しく展開されている例です．たとえば，「妥当な入力トランザクションを得る」機能は，「入力トランザクションを読取る」と「入力トランザクションの形式をチェックする」の2つの機能を総括した上位機能として表わされています．

このような正しい機能展開は問題の理解を大変容易にします．一方，図6.48は同じ問題を機能的にはとらえているのですが，かなり手順を意識した展開になっています．下位レベルの左から右へ順に実行していくことがくみとれます．

もちろん，この設計結果がまちがっている訳ではありません．手順的とは言っても，この結果を導き出すためには，おそらく，流れ図ではなくDFDで問題の分析を行なっているはずですから，個々の単位は機能的にとらえられています．
ただ，先の図6.47の結果と比較すれば，わかりやすさの点で先の結果に軍杯があがるのではないでしょうか．

また，後の結果は制御の広がりの面でも問題を提供しています．根モジュール「技術情報を検索する」は7つのモジュールを呼び出しています．7つ程度ならなんとかがまんの限界内ですが，これが10も20もになると大変です．

何が大変かと言うと，それを呼び出すための制御論理が複雑になることが予想されるからです．一般に，横に大きく広がりすぎている階層構造は，上位モジュールでの制御が大変になり好ましい形ではありません．

これは何もソフトウェアの設計だけの問題ではなく，何にでも当てはまりますね．たとえば，一人の管理者が何十人も何百人もの部下をもてば，きめ細かな管理はできませんね．企業が大きくなれば管理階層が増えてきます．同様に，ソフトウェアも規模が大きくなれ

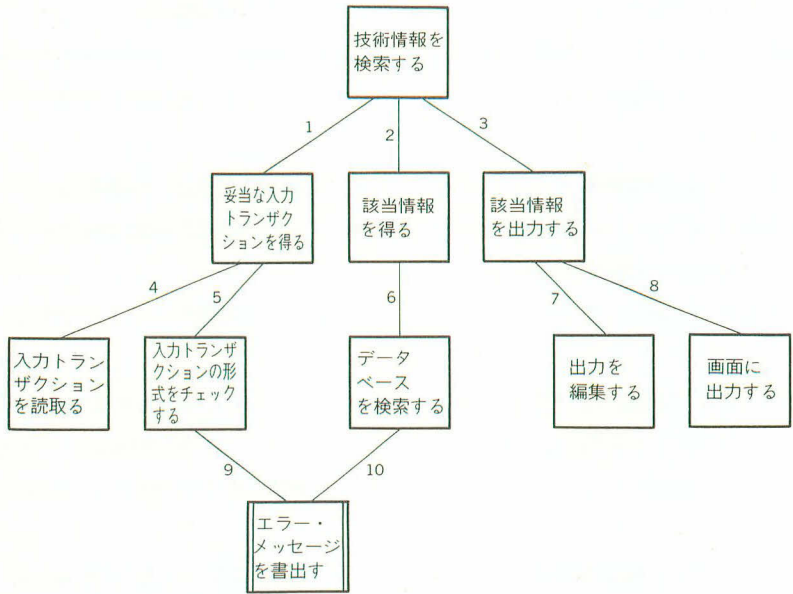
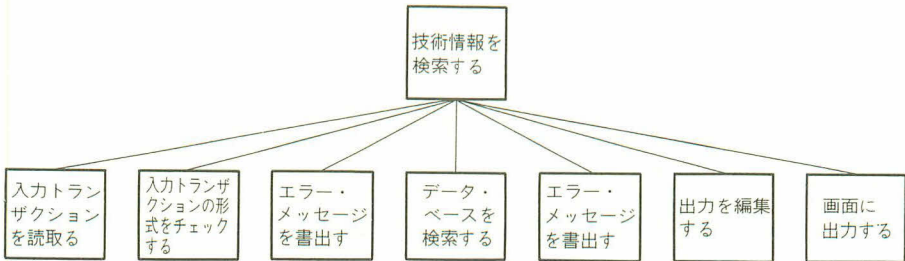
図6.47 「技術情報を検討する」の機能展開図

No.	入	出
1		妥当な入力トランザクション
2	妥当なトランザクション	該当情報
3	該当情報	
4		入力トランザクション
5	入力トランザクション	妥当な入力トランザクション
6	検索キー	該当情報
7	該当情報	編集済出力
8	編集済出力	
9	エラー・メッセージ	
10	エラー・メッセージ	

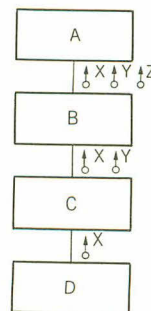
図6.48 「技術情報を検討する」のもう一つの機能展開図

ば階層が増えてくるのはごく自然のなりゆきです．

だからといって，必要以上に階層を深くするのも好ましくありません．たとえば，図6.49のように，たてに深く，横にせまい階層では，一般に，モジュール間のインターフェースに無駄が発生します．

この図で，Dの出力としてのXを最上位のAが必要とするときは，中間のB，Cは仲介の労をとる必要があり，本来必要でもないデータを引数として指定しなければならないことになります．

これは作業をわずらわしくするだけでなく，プログラムの理解を困難にし，エラーの入り込む可能性を大きくします．

このように，設計の結果を階層構造的にみて妥当かどうかをチェックしてみることが大切です．ただ，最も妥当な階層構造は一定の形におさまる訳ではありません．そこが設計の難しさでもあるのですが，最適構造は，あくまでも，問題構造に由来することをよく理解しておく必要があります．

図6.49
縦に深い階層構造ではB，CはデータXを仲介する必要がでてくる

▶入出力処理の隔離とインターフェースの簡潔さ

入出力処理，とくに特定ファイルの操作などを1つのモジュールにまとめる方向で検討するのは，つぎのような点で有利です．

- データ・セキュリティの配慮がしやすい．
- ファイルの変更や拡張などにたいし，そのモジュールだけ変更すればよい．
- 特定のデータ構造やレコード様式の処理を1つのモジュールに限定することにより，他のモジュール群は，レコード全体やデータ構造を受け渡しすることが少なくなり，必要な個々のフィールドを渡すだけでよくなる．結果として，モジュール間インターフェースが簡潔になる．
- 共通結合(共通域によるデータの受渡し)，スタンプ結合(パラメータでのデータ構造の受け渡し)が少なくなる．

これらは情報隠匿の概念として先に説明しましたね．

結果として，全体のモジュールが参照するデータ量の最小化をはかることができます．このばあい，ファイル操作を行うモジュールには，そのファイルを操作するいくつかの機能が含まれることになり，モジュールは情報的強度をもちます．

このような情報的強度をもったモジュールは，機能的強度をもったモジュールとくらべて優るとも劣らない価値をもつことになります．情報的強度モジュールは，最近注目されているオブジェクト指向のプログラミングの考え方にそったものであるとも言えます．

▶制御範囲と影響範囲

プログラムの設計結果を制御範囲と影響範囲の観点から評価しなければならないことに関しては第1章で座席を予約するプログラムの例をもとに詳述してあります．

第1章の該当部分をもう一度読みなおして，その重要性をもう一度確めて下さい．プログラム設計の本質が理解できたいまは，読みなおすことで，最初考えていたよりずっとその大切さを痛感されることと思います．

7．モジュールの論理設計とコーディングを行おう

前章でプログラムの構造化設計の基本的な手順は理解していただけたと思います．構造化設計は，プログラムを機能的な側面からとらえてその構造化をはかることが狙いでした．そして，機能構造をモジュール構造に対応させました．

プログラムのモジュール構造ができあがれば，つぎの設計作業は，個々のモジュールの論理の設計とその結果をもとにしたコーディングです．機能と論理の違いについては，第2章で説明しました．要は，機能は何を行うかであり，論理はどう行うかです．換言すれば，論理は手順であり，論理設計はモジュールで行うべき機能をどのような手順で行うかを設計することです．

この章では，まずモジュールの論理設計の方法について少し詳しく考えてみることにしましょう．そして，つぎにその結果をコーディングする方法について考えてみます．

7.1　構造化定理を用いてモジュールの論理を設計してみよう

構造化設計の狙いは，わかりやすいプログラムを作ることにありました．モジュールの論理を設計するさいに考慮すべきことも，やはりわかりやすさの具現化にあります．

わかりやすい論理を実現するためには，構造化設計のときと同様に，構造化の概念が必要になります．ただし，論理の構造化にさいしては，分割，独立性，階層化の考え方は必要ありません．かわりに，順次，選択，繰返しの3つの構造が必要になります．

このことを詳しく理解するためには，まず**構造化定理**について知っておく必要があります．構造化定理は，イタリアの科学者 Bohm と Jacopini によって最初に提唱されました．簡単に説明すればつぎのようになります．

図7.1　順次処理

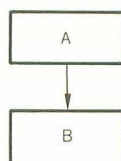
《構造化定理》

プログラムが1つの入口と1つの出口をもつように設計されていれば(このようなプログラムを適正プログラムと呼ぶ)，基本的な3つの制御構造の組合せによってどんな論理でも記述できる．3つの制御構造とは，**順次**，**選択**，**繰返し**構造のことである．

▶順次構造

それでは，この3つの制御構造，順次，選択，繰返しとはどんなものなのでしょうか．まず，順次(SEQUENCE)とは，文字どおり順番に実行する手順のことであり(図7.1)，モジュール内の命令あるいは部分機能を出てくる順に実行する型です．図7.2はそれを図示したものです．この図で，A，Bは1つの命令でもよいし，1つの部分機能と考えてもよいのです．A，Bを部分機能と考えれば，それを実行するための特定の論理があり，それは必ずしも順次にならないかもしれませんが，それはA，Bの中のことであり，AとBの単位でみれば，AのつぎにBが実行されるという意味で順次なのです．

図7.2　順次構造

```
┌─────┐
│  A  │
└──┬──┘
   ↓
┌─────┐
│  B  │
└─────┘
```

具体例でみてみましょう.

```
① sum=0.0;
② sum+=a;
③ sum+=b;
④ sum+=c;
⑤ mean=sum/3.0;
```

この例は, a, b, c の合計と平均を求めているC言語によるプログラムです. 手順そのものは, 特に説明するまでもないほど簡単なものですが, a, b, c の合計を変数 sum に, 平均を変数 mean に求めるために, ①から⑤までの命令が上から順に実行されます.

つぎも順次型の1つの例です. ただし, この例では, 命令ではなく, 機能を順次に実行しています.

① 正しいトランザクションを読み取る
② 該当レコードをデータベースから探す
③ 該当レコードを画面に出力する

「正しいトランザクションを読み取る」ためには, トランザクションを読取り, それが正しいかをチェックし, エラーがあればエラー・メッセージを書きだすといったより細かな機能を実行する必要があり, そのための論理は必ずしも順次ではありません (図7.3). また, つぎの「該当レコードをデータベースから探す」について考えてみても, 入力トランザクションが要求しているレコードをデータベースから見つけるための論理があり, そ

図7.3
「正しいトランザクションを読み取る」
の論理は順次と選択の組み合わせ

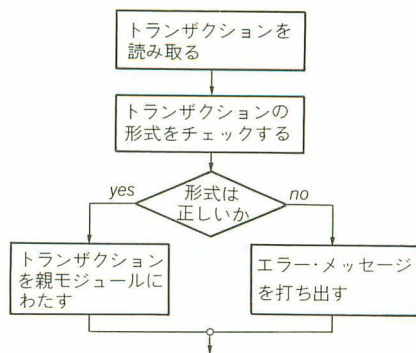
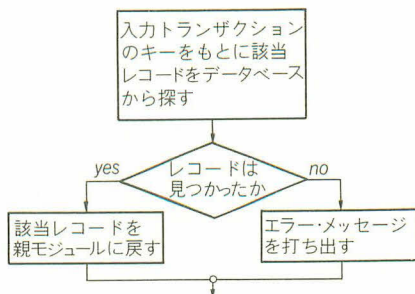



図7.4
「該当レコードをデータベースから探す」
の論理

れはもはや順次ではありません(図7.4).

しかし，①から③までの3つの機能単位でみれば，それらは順次に実行され結果として，入力トランザクションが要求したレコードが画面に出力されることになります.

▶選択構造(図7.5)

つぎに，選択構造(IF THEN ELSE)についてみてみましょう．選択構造とは，2つの異なった命令(または機能)の実行をある条件のテスト結果によって選択する型です．図7.6はその基本型を示しています．条件Pをテストし，その結果が真であればAを，偽であればBを実行します．ここで，A，Bは1つの命令でもよいし，1つの機能と考えてもよいのです.

図7.5 選択

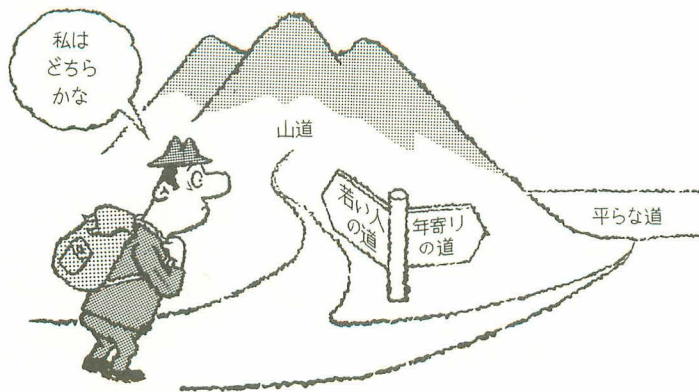



図7.6　選択構造の基本型

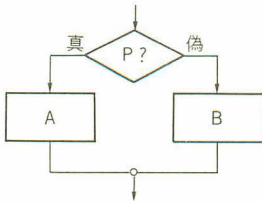
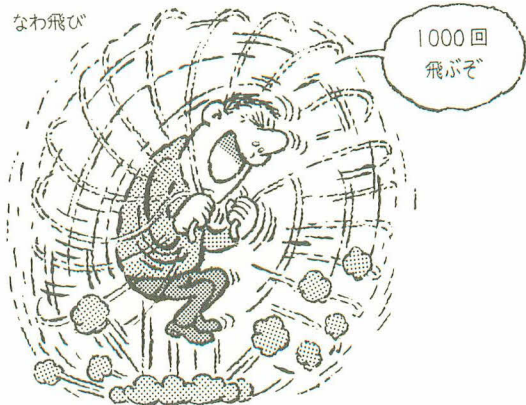



図7.7　繰り返し処理

選択構造の具体例を示します．

```
if(x<y)
  a=b;
else
  a=c;
```

この例では，条件としてxがyより小さいかをテストしています．そして，テスト結果が真(xがyより小さい)であれば

　　a=b

を実行します．一方，テスト結果が偽(xはyより小さくない)であれば

　　a=c

を実行します．

先に示した図7.3と図7.4の論理の後半部分が選択構造になっていることは，これまでの説明で十分理解できますね．

▶繰返し構造(図7.7)

3番目の構造は繰返し構造です．図7.8にその基本型を示します．条件Pが真である間，Aを繰返し実行します．Aが1つの命令でもよいし，1つの機能でもよいのは，順次，選択の場合と同じです．図7.8のような繰返し型は，一般にDO WHILE型と呼ばれています．

図7.8　繰り返し(DO WHILE)の基本型

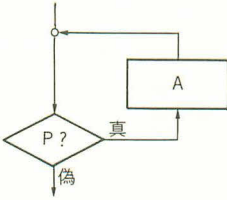
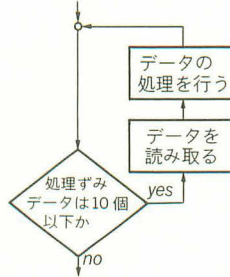
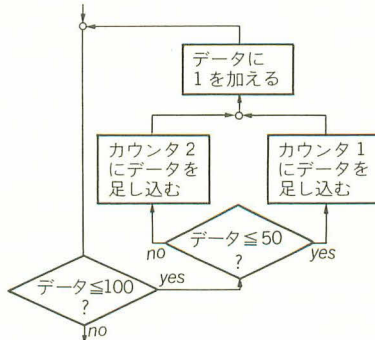



図7.9　繰り返し型の例

C言語のwhile文を使用すれば，この構造は容易に表現できます．

```
i=0;
while(i<5)
  {
    sum+=data [i] ;
    i++;
  }
```

上の例では，i<5がテストされる条件です．そして，この条件が満足される間(すなわち，iが5より小さい間)，{}でくくられた命令が繰返し実行されます．この例では，data [0] からdata [4] の値がsumにたし込まれていきます．また，繰返しが1度実行されるごとにiの値が1つずつ増していきます．

図7.10 「データを処理する」の詳細論理の例

C言語に強い方は，同じことがfor文を用いても表現できることをすでにご存知のはずです．

```
for(i=0;i<5;i++)
  sum+=data [i] ;
```

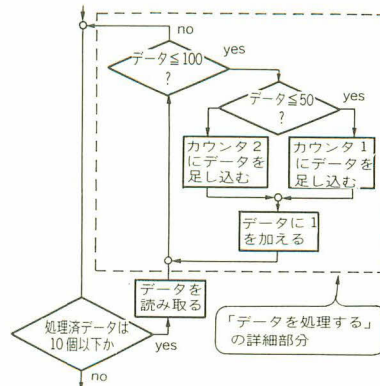
for文を用いれば，繰返しを制御する論理がfor文に集中でき，制御系と処理系を分解して考えることが可能になります．

▶3つの構造の組合せ

一般に，プログラムでは，複数個の入力データに対して同じ処理をほどこす型がよく現われます．その場合，論理が繰返し構造になることは容易に想像できますね．図7.9はそれを示しています．この例で，データの処理を行う部分をより詳細に分析していけば，その論理は順次，選択，繰返しの組合せで表現されることになるでしょう．たとえば，図7.10がその一例です．データの値が100以下である間，つぎのような処理を行っています．

- データの値が50以下であれば，カウンタ1にその値をたしこむ．
- データの値が50をこえれば，カウンタ2にその値をたしこむ．
- データに1を加える

そして，データの値が100を越えたとき，つぎの処理ステップに進みます．1つのデータを処理する論理が繰返し処理であり，その繰返し処理の中味が選択と順次からなりたっています(図7.11)．

図7.11 全体論理

このように，ある機能を処理する論理は，最終的には，順次，選択，繰返しの3つの基本構造の組合せで表現できることになります．

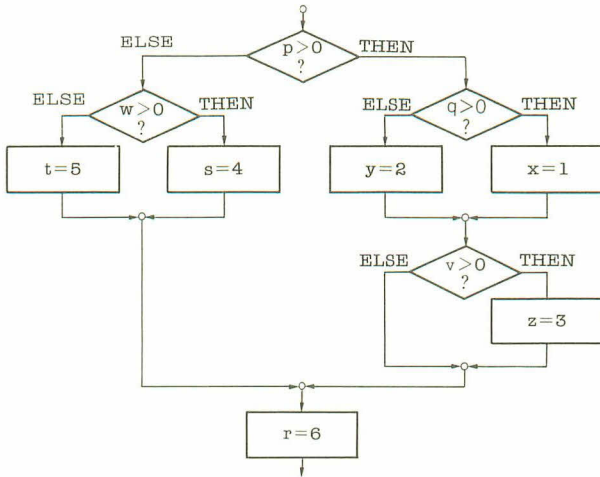
この例をC言語でコーディングすれば，つぎのようになるでしょう．

```
for(i=1;i<=10;i++)
    for(scanf("%d",& data ; data<=100;data++)
            if(data<=50)
                  cnt1+=data;
            else
                  cnt2+=data;
```

順次，選択，繰返しの3つの基本制御構造を用いれば，なぜ論理がわかりやすくなるのでしょうか．それはgoto文を使わないで論理を作成できる点にあります．

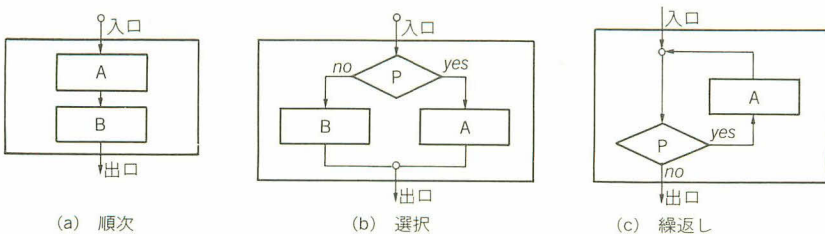
goto文を使って論理を作成した場合と使わないで作成した場合のプログラムのわかりやすさの比較は第1章ですでに見てきましたね．goto文の使用をやめてわかりやすくなった第1章の例を繰返しになりますが，ここでもう一度のせておきます．

```
if(p>0)
  {
      if(q>0)
          x=1;
      else
          y=2;
      if(v>0)
          z=3;
  }
else
  {
    if(w>0)
      s=4;
    else
      t=5;
  }
r=6;
```

図7.12 ネスティングした選択構造と順次構造の組み合わせ

再度ながめなおしてみれば，この例が選択と順次構造の組合せで作成されていることがわかります(図7.12).

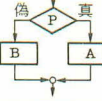
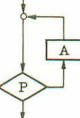
順次，選択，繰返しのどの構造も，それ自体1つの入口と1つの出口をもっていることにすでに気付いていられると思います(図7.13). 正規の出口以外から外部へ飛び出していくことはありません．したがって，これらの構造の組合せで作成した全体論理は，結果として，1つの入口と1つの出口を持つことになります．if 文や for 文を使っているかぎり，goto 文を使う必要はなくなります．

goto 文を使わないとプログラムがわかりやすくなる理由は，先の例でみられるように，原則的にプログラムを上から下へ順次に読んでいけるからです．

図7.13 順次，選択，繰り返しの3つの構造は1つの入り口と1つの出口を持っている

それによって思考の混乱が防げます．gotoがないがゆえに，z＝3を実行する条件は上から順にみていくだけですぐわかります．第1章の同じ論理をgotoを用いて作成した例ではこうはいきませんでしたね．それでも，この例は比較的小さなものですので，少し苦労すればgotoを使用していても正解にたどりつけますが，数十ページ，ときには数百ページにおよぶプログラム・リストではそうはいきません．

1ページ目に出てきたgotoによって50ページに飛んでいき，さらにそこで出てきたgotoで30ページ目に戻る…といったことに何度もぶつかると，たいていの人はソース・リストを投出してしまうのではないでしょうか．

このように，論理を作成するときに構造化定理をもとに行えばgoto文の使用を避けることができ，それによるメリットを享受することができるのですが，そのことにこだわりすぎると逆効果をまねくこともあります．ときには，gotoを用いた方が論理がわかりやすくなることもあります．たとえば，

(1) モジュールの終りの部分へのgoto．
(2) 異常終了時のループからの脱出．
(3) gotoを使わないようにすることによって，モジュールのいくつかの部分に同じコード・パターンがあらわれてしまうようなとき．

などです．

gotoのわずらわしさは，論理のどの部分へも自由に飛んでいけるところにあるわけですが，飛んでいった先で論理が終了する(仕事が終わる)なら思考の混乱もなく，わかりやすさをそこねることはありません．このことは異常終了時の処置にもあてはまります．C言語では，break文がその役目をはたしていましたね．

また，goto文の使用を避けるために生じた論理の重複は，将来の変更時にエラーの発生度合を高める原因になります．

論理を作成するときに最も配慮しなければならないことは，そのわかりやすさを実現することにあります．gotoを使わないようにすることにあるわけではないのです．このことを誤ちがえないようにして，gotoの使用，未使用を慎重に検討していく必要があります．図7.14に基本制御構造のまとめを示しておきます．

図7.14　基本制御構造のまとめ

基本制御構造	流れ図	C＋＋言語
順次	A → B	A; B;

選択	偽 P 真 / B A	if(P)
		A;
		else
		B;

繰返し	P / A	while(P)
		A;
		または
		for (文1;条件;文2;)
		A;
		(注)
		文1:制御変数の初期化
		文2:制御変数の増分

7.2　論理のわかりやすい文書化を心がけよう

わかりやすい論理を実現させるためには，構造化定理にもとずき，順次，選択，繰返しの3つの制御構造の組合せで論理を表現する必要があることは理解できた思います．

このように作成した論理を誰にでも理解できるように文書化しておくのも大切なことです．

▶流れ図

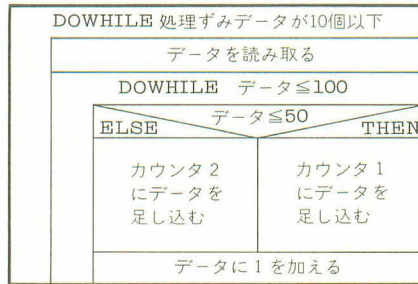
論理を文書化するとき，一番多く用いられてきたのは流れ図です．本書でも，これまでの部分ですべて流れ図を用いて説明してきました．

流れ図は表現方法に自由度が大きく，どんな論理でも表現可能です．しかし，逆に，その自由度の大きさが構造化論理の表現には好ましくない結果をもたすことがよくあります．

流れ図上での矢印はどの部分にものばすことができます．これは構造化をくずす原因になります．最初は慎重に順次，選択，繰返し構造だけで論理を組立てていても，ちょっとした変更で，構造化をくずしてしまうような箇所へ矢印をのばしてしまいます．

また，少しこみいった論理の構造化を流れ図で表現したばあい，論理の構造を一見しただけで簡単に識別することが難しくなります．図7.3や図7.4の例のようなときは簡単に識別できますが，図7 11程度になってくるとやや識別しにくくなってくることがわかると思います．現実のプログラムの論理は，これらよりもずっと複雑です．それらの構造化を流れ図で表現するには無理があります．

図7.15 NS チャートによる論理表現

DOWHILE 処理ずみデータが10個以下
データを読み取る
DOWHILE データ≦100
データ≦50
ELSE
THEN
カウンタ2にデータを足し込む
カウンタ1にデータを足し込む
データに1を加える

流れ図は，基本的に，構造化指向の文書化手法ではありません．

▶ NS チャート

そこで，構造化論理を理解しやすい形で文書化するために，流れ図にかわって，NS チャート(Nassi-Shnei-derman Chart)が用いられることが多くなっています．

図7.15を見て下さい．これは，図7.11の流れ図で表現したと同じ内容のものを NS チャートで表現したものです．NS チャートでは，順次，選択，繰返しに対応した表現方法しかもっていません．図7.16にその基本型が示されています．

順次構造は，順次に処理するものを矢印なしに個々の枠内に表現します(図7.16(a))．

選択構造は，条件部分を頭部に記述し，テスト結果に応じた処理部分をその下部にならべて記述します(図7.16(b))．

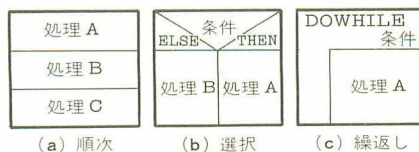
繰返し構造は，繰返し条件を頭部に記述し，処理部分をその下部に記述します(図7.16(c))．

図7.15は，この3つの基本型の組合せで表現されていることが理解できると思います．一番外側に繰返し構造が出てきます．この部分は10個のデータに対し1個，1個その下

図7.16 NS チャートの基本表現

処理 A
処理 B
処理 C
(a) 順次

条件
ELSE
THEN
処理 B
処理 A
(b) 選択

DOWHILE 条件
処理 A
(c) 繰返し

部の部分の処理を繰返します．この繰返し処理の部分は，データを読取る処理とデータ値が100になるまでの繰返し処理との順次構造になっています．さらに，データが100になるまでの繰返し処理は，データ値が50以下かをテストしている選択構造とデータに1を加える処理の順次構造になっています．

このように，NSチャートでは，流れ図で用いたような処理の流れを示す矢印は用いていません．3つの構造に対応した記法を組合せていくことで，自然に処理手順も表現しています．

構造に対応した記法を用いることで，論理構造が容易に識別できます．それによって，論理径路が明確になりテスト・ケースを設計するときも大変やりやすくテストも経路をなくすことができます．

また，構造に対応した記法しか用いていませんので，あとで変更が生じたときも，流れ図のように構造化をくずしてしまうおそれはありません．

最近はマイコンの性能向上にともない，マイコン用に作成するプログラムの規模も大きくなってきています．したがって，いままでのように1人ですべて作成するのではなく，多人数で1つのプログラムを手分けして作成する機会もふえています．

そのようなとき，お互いの意思疎通をはかり，かつ保守のためにもわかりやすい文書化は欠せなくなってきています．NSチャートはそのための大変有効な手法です．

7.3 疑似コードを用いてコーディングの準備をしよう

「疑似コード」は構造化設計によりモジュール化したあと，モジュールの論理設計を行う段階で用いられます．モジュールが行うべき機能の論理を頭に思い浮かべて，一気にプログラミング言語(C言語やFortran)でコーディングできるなら，それはそれで効率的です．しかし相当の経験をつまないと最初からコーディングして構造化された論理で，間違いなく作成することは容易なことではありません．

そこで，コーディングする前に一度疑似コードを用いて，論理を整理してみて自信ができてからコーディングすると，それだけソフトウェアとしての信頼性が高くなるというわけです．

疑似コードはその名前が示すように，C言語やFortran言語のように現実に製品として存在するものではありません．したがって，言語固有の文法上の制約もなく，表現は自分の好きなようにします．自分がいちばんわかりやすい表現を使えばよいのです．

ただ，最低限の約束はむしろあったほうが書きやすい面がありますので，通常はつぎのような約束のもとに疑似コードを使用します．

- 論理を組み立てるときは，構造化定理を採用する．
- 選択や繰返しの表現には，IF，THEN，ELSE，ENDIF，DOWNHILE，ENDDO などのキー用語を用いて，その部分が選択，繰返しであることが明確にする．
- 選択や繰返しがネスト(入れ子)するときは，その階層がわかるようにレベルごとの書出し部分をずらす(字下げ(インデント)とよばれている)．
- 単一機能は1行に書き，別の機能の記述は行をあらためる．

具体例でみてみましょう．

```
DOWHILE　処理済データが10個以下である
  データを読取る
  DOWHILE　データ値が100以下である
    IF　データ値が50以下である　THEN
            カウンタ1にデータ値を加える
    ELSE
            カウンタ2にデータ値を加える
    ENDIF
      データ値に1を加える
  ENDDO
    処理済データ個数に1を加える
ENDDO
```

これは図7.15のNSチャートで表現した論理と同じものを疑似言語で示したものです．

ここでは2つの繰返し論理(DO WHILE)がネストしています．外側の繰返し処理は入力データ10個のそれぞれに対し行なわれます．内側の繰返し処理は個々の入力データに対し，そのデータ値が100以下の間で行われます．

この内側の繰返し処理の内容は，選択処理(IF THEN ELSE)になっています．

これらの繰返し処理や選択処理は，それぞれ頭にDO WHILE，IF THEN ELSEのキー用語を用い，制御範囲の終りには，ENDDO，ENDIFのキー用語を用いています．また，字下げのルールを併用することで，それぞれの制御範囲がより一層明確になり，論理

の読みやすさが増しています.

疑似コードは，NS チャートと同様に，プログラムの文書化に有効です．NS チャートが図的であるのに対し，疑似コードは文章的で，プログラミング言語的表現に近くなっています.

文書化として NS チャートを用いるか，疑似コードを用いるかはそれぞれの特徴があり，意見のわかれるところですが，本書では，よりコーディングに近い形式であり，モジュールの関係が明確に表現できる利点をとり，この後は疑似コードにより，論理を表現していくことにします.

7.4 モジュールのコーディングを行おう

プログラムの設計作業は，基本的には特定のプログラミング言語とは独立しています．すなわち，プログラムの問題仕様を分析し，モジュール構造を設計したり，個々のモジュールの論理の構造化を考えたりする時，使用するプログラミング言語とは関係なく作業をすすめることができます.

しかし，NS チャートや擬似言語で論理を設計したあと．いよいよモジュールごとのコーディング作業に入る段階になると，当然のことながら，使用するプログラミング言語を考慮に入れる必要があります.

構造化定理に準じた論理にそってコーディングを行う場合，プログラミング言語によって適，不適があるのは事実です．たとえば，大規模商用プログラム作成のときによく使用される COBOL 言語は，順次や選択構造は比較的素直に表現できますが，繰返し構造の表現に難色がありました．技術計算によく使用される FORTRAN 言語も選択や繰返し構造の表現に一工夫が必要でした．COBOL や FOTRAN は構造化の必要性が認識される以前に出現した言語ですので，構造化の思想が反映されていなかったのも止むを得なかったのです．しかし，これらの言語も構造化技法の普及にともない，次第に構造化にそった改良が行われるようになりました.

その点，本書が対象にしているマイコン用のプログラムには，主として C 言語が使われることが多く，これらの言語は比較的新しい時代に出現したこともあり，構造化論理の表現に大変便利な命令をそなえています.

順次，選択，繰返し構造に対する C 言語の表現方法は先述したとおりですが，確認の意味で図7.14 をもう一度みておいて下さい.

7.5　例題をもとに考えてみよう

構造化コーディングについて理解できたところで，簡単な例題をもとに，構造化設計から構造化コーディングに進めていく様子を学ぶことにしましょう．

例題1：三角形の形を決める(図7.17)

三角形の三辺A，B，Cの大きさを入力として読取る．読取ったデータをもとに三角形の形を決めて，つぎの該当コードを出力する．

コード	三角形の形
0	三角形でない
1	2～5の形に属さないもの
2	2辺のみ等しい
3	すべての3辺は等しい(正三角形)
4	直角三角形
5	直角三角形で，2辺は等しい

▶問題の分析

この問題のポイントは，題名が示すように，三辺のデータをもとに三角形の形を決めるところにあります．

三角形の形を判定するためには，つぎのような論理が考えられます．

(1) まず，読取った3つのデータが三角形を構成する条件を満足するかどうかを調べます．三角形であるためには，2辺の和が他の一辺より大きいことが必要です．すなわち，

$$A+B>C$$

のときは三角形を構成し，

$$A+B\leqq C$$

のときは三角形にはなりません．

(2) 三角形の条件を満たすことがわかれば，次はいよいよ三角形の形を判定することになります．まず，ピタゴラスの定理で直角三角形であるかどうかを判定できるのは誰でも気

図7.17 三角形の形を決める

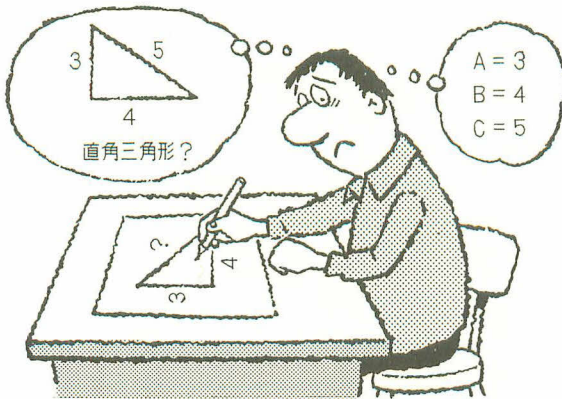
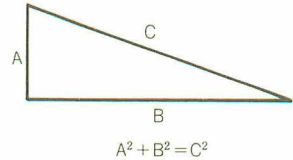



図7.18 ピタゴラスの定理による直角三角形の判定

がつくことでしょう(図7.18).

$$A^2+B^2=C^2$$

これらの判定のためには，三辺のデータをあらかじめ昇順に分類しておく必要があります.

(3) 直角三角形であることがわかれば，つぎに2等辺直角三角形か否かを判定します.
三辺のデータが昇順にA，B，Cと分類されているなら，

$$A=B$$

なら2等辺直角三角形であり

$$A \neq B$$

ならただの直角三角形ということになります.

(4) 直角三角形でないときは，正三角形か2等辺三角形かそれ以外の三角形かを判定します.

$$A=C$$

であれば2等辺三角形

$$A=B=C$$

であれば正三角形，そうでないときはただの三角形ということになります.

▶構造化設計

問題自体の構造は簡単です．問題仕様からみて，この問題は源泉/変換/吸収分割でごく簡単に構造を決定できます．この問題の主要な機能は，

⑴ 入力データ(3辺のデータ)を読取ること

⑵ 読取ったデータを昇順に分類すること

⑶ ⑵の結果を用いて，三角形の形を判定すること

⑷ 判定結果を出力すること

です．これをデータ・フロー・ダイヤグラムで表現すると図7.19のようになるでしょう．

この図から，モジュール構造とモジュール間インターフェースも容易に決定できます．その結果を図7.20に示します．

図7.19のデータ・フロー・ダイヤグラムは，図7.20のような結果ではなく，むしろ図7.21のような結果を期待した方も多いと思います．最大抽象入力点の位置からみて，源泉モジュールは「分類ずみ3辺のデータを得る」にし，その下に読取りモジュールと分類モジュールをぶらさげるのが妥当でないかという考え方があるからです．

もちろん，この考え方も間違いではありません．分類するという機能を入力機能の一部として把えるか，三角形の形を判定する機能の一部として考えるかの違いだけで，どちらの結果も，この問題に関する限り，コーディングや保守のやりやすさの観点からはさほどの違いはありません．

ただ，分類機能を入力機能の一部に入れることに対しやや異和感がある気がします．この問題では，入力データを分類する必要がありますが，一般的には，その必要のないことの方が多いでしょう．分類するのは，あくまでも，三角形の形の判定のために必要なのであって，入力機能の本質ではありません．

機能的にみる限り，図7.21の構造図の方がすきっりしています．これはモジュールの汎用性の面から重要なことです．入力機能モジュールを他のプログラムで活用しようとしたとき，分類機能が冗長になってしまう可能性があるのです．

その意味で，この例題の構造図は図7.21の方を採用することにします．

▶モジュールの論理設計

図7.21のモジュール構造にもとずいて，この問題のキーになるモジュールの論理を疑似コードを用いて設計してみましょう．

まず，主モジュール「三角形の形を決める」の論理ははつぎのようになります．

図7.19 「三角形の形を決める」問題のデータ・フロー・ダイヤグラム

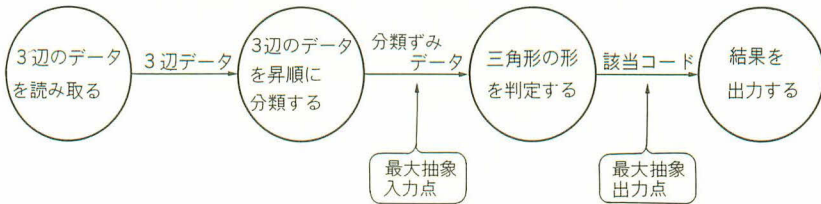
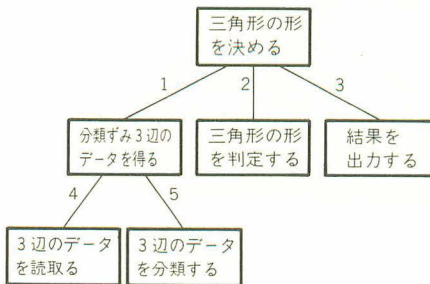
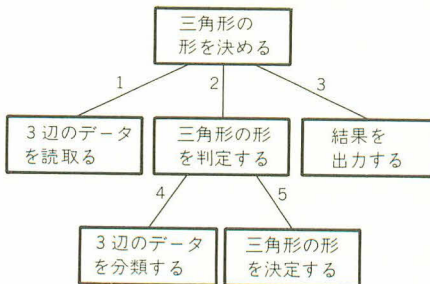



図7.20 「三角形の形を決める」問題の構造化

インターフェース表

番号	入	出
1	—	分類ずみデータ (A, B, C)
2	分類ずみデータ	該当コード
3	該当コード	—
4	—	3辺のデータ
5	3辺のデータ	分類ずみデータ

図7.21 「三角形の形を決める」問題のもう一つの構造図

インターフェース表

番号	入	出
	—	3辺のデータ (A, B, C)
2	3辺のデータ	該当コード
3	該当コード	—
4	3辺のデータ	分類ずみデータ
5	分類ずみデータ	該当コード

```
三角形の形を決める(MAIN)
DOWHILE(MORE-DATA)
  3辺のデータを読取る
  三角形の形を判定する
  結果を出力する
ENDDO
END MAIN
```

一般に，主モジュールの論理は，従属モジュールを呼出す制御論理が中心になります．

つぎに，「三角形の形を判定する」モジュールについて考えてみます．このモジュール自体は，従属するモジュールを呼出すだけのごく簡単な論理になります．

```
三角形の形を判定する
3辺のデータを分類する
三角形の形を決定する
END
```

「3辺のデータを分類する」モジュールの論理は，いわゆる分類のための論理になり，いろいろな方法がありますが，その1つをつぎに示しておきます．論理をよく吟味して，3辺のデータが昇順に分類される様子をよく確かめてみて下さい．結果は

A＜B＜C

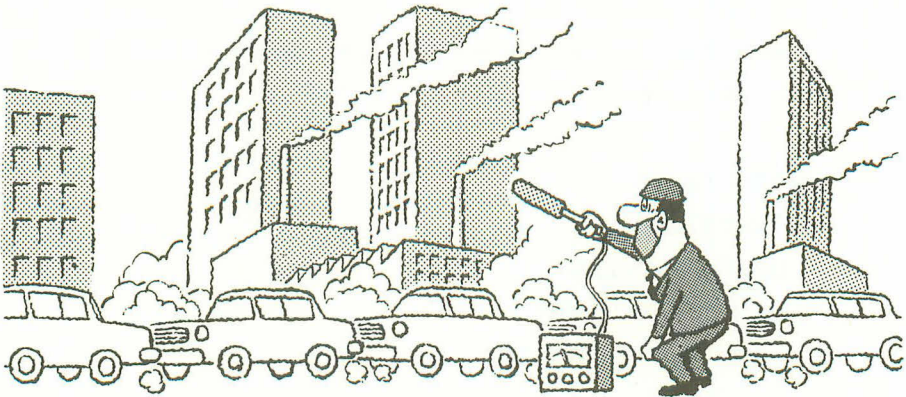
になります．

```
3辺のデータを分類する
K=2
DO I=1 TO K
  DO J=I+1 TO K+1
    IF SIDE(I)>SIDE(J) THEN
      DO HYPO=SIDE(I)
          SIDE(I)=SIDE(J)
          SIDE(J)=HYPO
      ENDDO
    ENDIF
  ENDDO
ENDDO
A=SIDE(1)
B=SIDE(2)
C=SIDE(3)
END
```

この論理の前提として，読取られた3辺のデータがSIDE(1)～SIDE(3)に記憶されているものとしています．

さて，最後に一番核になる三角形の形を決定する論理について考えてみましょう．基本的な考え方は，問題の分析のところで説明したとおりです．

```
三角形の形を決定する
IF A+B>C THEN
    IF A²+B²=C² THEN
        IF A=B THEN
            二等辺直角三角形である(コード5)
        ELSE
            直角三角形である(コード4)
        ENDIF
    ELSE
        IF A=C THEN
            正三角形である(コード3)
        ELSE
            IF A=B または B=C THEN
                二等辺三角形である(コード2)
            ELSE
                単なる三角形である(コード1)
            ENDIF
        ENDIF
    ENDIF
ELSE
    三角形でない(コード0)
ENDIF
END
```

図7.22 大気汚染度を測定する

以上説明したモジュールの論理が，順次，選択，繰返しの3つの基本制御構造からなりたっていることに注意して下さい．

例題2：大気汚染度の測定(図7.22)

もう1つの例題で考えてみましょう．第4章で一部とりあげた大気汚染度の測定問題についてまとまった形で検討してみることにします．問題の仕様は実際よりはずっと簡略化してあります．本書の目的は問題をより実際的に理解することではなく，構造化設計を理解することにあるわけですから，これで十分なのです．

▶問題仕様

《大気汚染度の測定》

規模の大きな製造工場の煙突の近くで24時間にわたって大気の汚染度を1分ごとに測定した．汚染度の許容測定範囲は1 ppmから180,000 ppmまでの間である．プログラムとして行うべき機能は，つぎのとおりである．

(1) 測定値は，キーボードから入力する．

(2) 測定値は数値化してあり，すべての値は計算に使える．その範囲は1～180,000 ppmであり，ゼロの値があれば，それは測定装置の誤動作を示す．

(3) 24時間のうち誤動作が起こらなかった各時間ごとの汚染度の平均値を計算する．

図7.23 「大気汚染度の測定」問題の印刷出力

時間	ppm（平均）
1	80864
2	86681
3	87540
4	75068
5	90222
⋮	⋮
20	88664
21	89657
22	91011
23	81726
24	92216

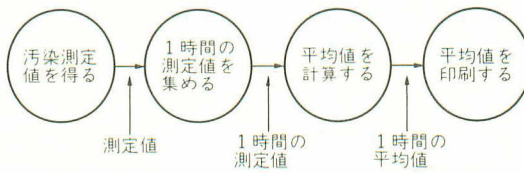
図7.24 「大気汚染度の測定」問題の大要

(4) 装置が誤動作を起こした時間帯では，その時間の ppm の平均値を 0 ppm にセットし，その時間に対するすべての処理は行わない．

(5) 1時間ごとの平均値は，図7.23のような形式で印刷出力する．

▶問題の構造化設計

さて，問題の分析にとりかかりましょう．分析の第一歩は，問題構造の大要をつかむことです．解くべき問題を，たとえば3～10個くらいの部分機能で表現してみます．部分機能をとりあげるときは，問題をデータの流れにそって分析し，主要なものをとりあげるようにします．この問題では，「汚染測定値を得る」，「1時間の測定値を集める」，「1時間の平均値を計算する」，「平均値を印刷する」といったところが主要機能になるでしょう(図7.24)．

図7.25 主要な入力，出力データの流れにそって部分機能をならべる

分析の第2のステップは，問題のなかの主要な入力と出力データの流れを明確にし，それらにそって最初にとりあげた部分機能をならべてみることです．

例題では，図7.25のようになります．これは第4章で説明したデータ・フロー・ダイヤグラムそのものに他ありません．

分析の第3ステップは，問題構造のなかで入力と出力に対する最大抽象点を見つけることです．

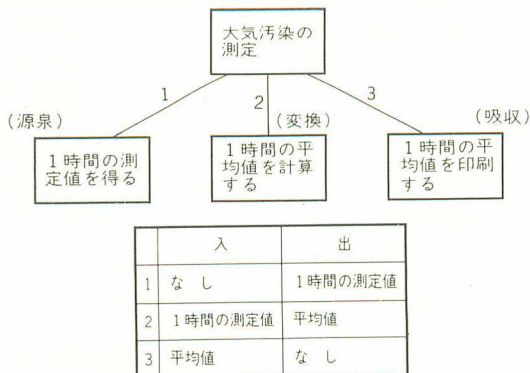
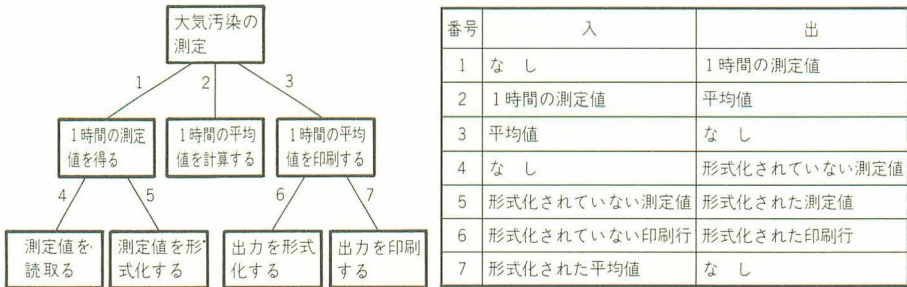
最大抽象入力点を見つけるには，第2のステップでならべた問題構造の出発点から始めて，入力の流れを見つめながら問題構造のなかに入っていくと，入力データがどんどん抽象化されて，もはやこれ以上は入力とみなせなくなる点に達するはずです．そこが最大抽象入力点になります．

一方，最大抽象出力点を見つけるには，逆に問題構造の終点から出発し，出力の流れを見つめながら問題構造のなかをさかのぼっていくと，出力はだんだん抽象化された形にな

図7.26 「大気汚染度の測定」のモジュールへの展開

	入	出
1	なし	1時間の測定値
2	1時間の測定値	平均値
3	平均値	なし

図7.27 「大気汚染度の測定問題」の最終結果

番号	入	出
1	な　し	1時間の測定値
2	1時間の測定値	平均値
3	平均値	な　し
4	な　し	形式化されていない測定値
5	形式化されていない測定値	形式化された測定値
6	形式化されていない印刷行	形式化された印刷行
7	形式化された平均値	な　し

ってきて，ついには出力の流れが最初に現れる点に到達します．ここが最大抽象出力点です．

このことを例題で考えてみましょう．最大抽象入力点は，「1時間の測定値を集める」機能と「平均値を計算する」機能の間になります．その理由は，図7.25の問題構造を開始点からみていけば，「汚染測定値を得る」と「1時間の測定値を集める」は入力データの流れとしてとらえることができるのに対し，「平均値を計算する」は出力を求めるための処理であるからです．

一方，最大抽象出力点は「平均値を計算する」と「平均値を印刷する」の間になります．「平均値を印刷する」は，あきらかに出力の処理であり，その出力は「平均値を計算する」機能が行われた後にはじめて姿を現します．

これら二つの最大抽象点を見つけ出せば，問題を三つのもっとも独立した部分機能，すなわち源泉(入力)，変換(入力から出力へ)，吸収(出力)機能に分割できることになります．

結果を図7.26に示します．源泉，変換，吸収機能が単一機能として表現できている(一つの目的語と述語)点に注意して下さい．

ここまで分析がすすめば，後は，源泉，変換，吸収機能をさらに分割する必要があるかどうかを検討します．分割の必要があると認められれば，それまでと同じやり方で分割していきます．分割の必要性に対する判断基準は，機能をコーディングするときにその論理が頭に思い描けるかどうかです．

例題では，理解を容易にするために問題を簡易化しているので，これ以上の分割は通常必要ありませんが，分割という概念をよりよく理解するために，あえて分割をつづけてあります．分割の最終結果を図7.27に示します．

▶モジュールの論理設計

図7.26のモジュール構造にそって主要なモジュールの論理を疑似コードによって設計してみましょう.

まず，主モジュールはつぎのようになります.

```
大気汚染を測定する.
    時間を1にセットする.
    DOWHILE 時間が25未満の間.
        1時間の測定値を得る.
        平均値を計算する.
        平均値を印刷する.
        時間に1を加える.
    ENDDO
END
```

主モジュールに従属している3つのモジュール「1時間の測定値を得る」，「平均値を計算する」，「平均値を印刷する」が1時間ごとに実行されることになるでしょう．したがって，この3つのモジュールを呼び出す繰返し構造が現れています．DOWHILE, ENDDOという用語を用いることによって，繰返し構造の制御範囲が大変明確になっている点に注目して下さい．また，字下げのルールにしたがって記述されていることが，より論理を明解にしている点にも注目して下さい.

次に，「1時間の測定値を得る」モジュールの論理は次のようになるでしょう.

```
1時間の測定値を得る.
    分を1にセットする.
    DOWHILE 分が61未満の間.
        測定値をテーブルに読み取る.
        分に1を加える.
    ENDDO
END
```

ここでも，１分ごとの測定値を１時間分読み取るために繰返し論理を設計しています．「平均値を計算する」モジュールの論理は次のようになります．

```
平均値を計算する.
    合計値を0にセットする.
    分を1にセットする.
    DOWHILE 分が61未満の間.
        IF  テーブル値が0
                合計域を0にセットする.
                break
        ELSE
                合計域にテーブル値(測定値)を加える.
    ENDDO
    合計値を60で割り平均値を求める.
END
```

最後に，「平均値を印刷する」モジュールの論理は，ごく簡単に次のように設計しておきます．

```
平均値を印刷する.
    時間と平均値を印刷する.
END
```

▶モジュールのコーディング

C言語によるモジュールのコーディング例を参考までに示しておきます．擬似コードからC言語に変換する作業はそれほど難しくありません．7.4で述べたように，C言語は構造化論理を表現するのに適した命令文をもっています．

主モジュール「大気汚染度を測定する」

```
#define datasize 60
#define endHour 24
main()
{
  long ppm [datasize] ;
  int hour;
  double mean;
  for(hour=1;hour<=endHour;hour++)
    {
      getdata(ppm);
      mean=cmean(datasize,PPm);
      putmean(hour,mean);
    }
}
```

「1時間の測定値を得る」

```
#define endMin 60
getdata(ppm)
long ppm [] ;
{
  int minute;
  for(minute=0;minute<endMin;minute++)
    scanf("%d",ppm [minute] );
}
```

```
「平均値を計算する」
cmean(datasize,ppm)
int datasize;
long ppm [ ] ;
{
  double sum;
  sum=0.0;
  for(i=0;i<datasize;i++)
    {
      if(ppm [i] ==0)
        {
          sum=0.0;
          break ;
        }
      else
          sumt=ppm [i] ;
    }
  return(sum/datasize);
}

「平均値を印刷する」
putmean(hour,mean)
  int hour;
  double mean;
  {
    printf("%4d %6.0f/n",hour,mean);
  }
```

主モジュール「大気汚染度を測定する」は，3つの直接従属モジュール「1時間の測定値を得る」，「平均値を計算する」，「平均値を印刷する」をもっています．このコーディング例では，これら3つのモジュールをそれぞれ getdata，cmean，putmean という名前の関数として扱っています．主モジュールの論理はこれら3つの関数を1時間ごとに呼

出す繰返し構造からなりたっています．

繰返し構造はfor文で表現しています．この箇所はwhile文でも表現できますが，制御情報と処理情報をはっきり区別けして表現できる点を評価してfor文を採用しています．これによって，モジュール論理がわかりやすくなります．

処理の基本データである測定された大気汚染度データはppmという名前のテーブルに貯えます．ppmは配列で60個のデータを貯えるようになっています(図7.28)．1分ごとに測定されたデータを60個貯えることにより，テーブル全体で1時間分のデータを貯えます．

各モジュールのコーディング論理が先述の疑似コードの論理にそっていることを確めて下さい．また，{}や字下げのルールで繰返しや選択の制御範囲を明確にしている点にも注意して下さい．

各モジュールを強度的にみれば，それぞれ機能的強度になっています．また，結合度はデータ結合になっています．すなわち，モジュール間でのデータのやりとりは引数として指定しています．これにより，モジュール間でどんなデータのやりとりがあるかがはっきりわかり，プログラムの理解度が向上します．

たとえば，

```
getdata(ppm)
```

から，大部分の人は，このモジュール(関数)が入力データを読取り，それをテーブルppmに貯えることを理解できるでしょう．また，

```
cmean(datasize, ppm)
```

から，このモジュールがppmのdatasize個のデータ(この場合は60個)の平均を計算することを推察するのはそれほど難しくはないでしょう．

図7.28　一時間の測定値を貯えるテーブル

ppm [60]

[0]	入力データ1
[1]	入力データ2
	⋮
[58]	入力データ59
[59]	入力データ60

もちろん，変数名や関数名をつけるとき，内容をほうふつさせるような名前にするよう心がけることも理解度を向上させる上で大変重要なことです．

また，適切なコメントをモジュールの文に付加することで理解度はさらに向上するでしょう．たとえば，先に示した主モジュールのコーディング例に以下のようなコメントを追加すれば，わかりやすさとしては万全になります．

```
#define datasize 60   /*汚染度テーブルの大きさ*/
#define endHour 24   /*測定時間の上限*/
main()        /*大気汚染度を測定する*/
{
  long ppm [datasize] ;        /*汚染度テーブル*/
  int hour;                    /*時間*/
  double mean;                 /*平均*/
  for(hour=1;hour<=endHour;hour++)
    {
      getdata(ppm);                  /*1時間の測定値を得る*/
      mean=cmean(datasize,ppm);      /*平均値を計算する*/
      putmean=(hour,maen);           /*平均値を印刷する*/
    }
}
```

コーディングを工夫することで，モジュール間のデータの受渡しの数をできるだけ少なくすることはできます．たとえば，C言語の場合，上の例で，モジュール間の引数として使用されている ppm，datasize，hour，mean を外部変数として宣言すれば，モジュールの呼出しごとにいちいち数として指定する必要はなくなります．

その場合の主モジュールは次のようになるでしょう．

```
#define datasize 60
#define endHour 24
long ppm [datasize] ;
int datasize,hour;
double mean;
```

```
main( )
{
  extern long ppm [ ] ;
  extern int datasize hour;
  extern double mean;
  for(hour=1;hour<=endHour;hour++)
   {
       getdata( );
       mean=Mean( );
       putmean( );
   }
}
```

ご存知のとおり，データを外部変数として宣言すれば，main とそのなかで使用される関数(モジュール)で共有して使用できるようになります．

これらは結合度としてみれば，共通結合になります．第5章で検討したように，共通結合には注目すべきいくつかの特徴があります．まず，共有データを使用するとき，モジュール間で引数としていちいち指定する必要がないことです．この例でも，ppm は，それを使用する関数が主モジュールで呼出されても，引数としては表わされていません．

このことは，コーディング時にいちいち引数を書く必要がなく，コーディングの手間が省けることを意味しています．また，関数を呼出し，実行するときのオーバヘッド時間が少なくてすみます．これらの理由から，現実には，よくこの手のテクニックが使用されます．

このような利点はそれなりに価値がありますが，反面，問題点もあることに気付いて下さい．結合度はデータ結合にするのが最も好ましいことはすでに述べました．その主な理由は，データ結合にすれば，相手のモジュールをブラック・ボックスとして扱え，変更の波及効果を最小にできることと，データを引数として指定することにより，モジュール間でどんなデータのやりとりがあるかをはっきり知ることができ，それが結果としてプログラムの理解度の向上につながる点にありました．

使用するデータを引数として指定しない場合は，たしかにコーディングの手間は省けますが，わかりやすさをそこねます．先のコーディングを初めて見た人は

```
getdata( );
```

という文をみても，その名前からこの関数が入力データを読取ることを推察できても，読取ったデータをどこに貯えるかまではわかりません．

その意味で，

```
getdata(ppm);
```

としておけば，引数として指定されたppmに入力データが貯えられると推察することができるでしょう．

また，

```
mean=cmean( );
```

では，関数Meanがその名前から平均を計算することはわかっても，何の平均を計算するのかはmainをみただけではわかりません．しかし，先述したように，

```
mean=Mean(datasize, ppm);
```

としておけば，配列ppmのdatasize個のデータの平均を計算することが先の例よりずっと容易に推察できることになります．

このように，引数を指定することにより，プログラムのわかりやすさを向上させる効果はありますが，気を付けたいのは，引数が多くなりすぎると逆効果になってしまうことです．

たとえば，

```
getdata(a, b, c, d, e, f, g. );
```

のような文に出合ったとき，7個の引数が何を意味するかを理解するのが大変になり，かえって，プログラムのわかりやすさをそこねます．また，コーディング時に引数の指定ミスといったエラーを誘発しかねません．

引数の数は何個が最適なのかは，その関数の処理内容とのからみもあり一概に決めつけるわけにはいきませんが，一般的には，5個以内程度に収めるのがよいと思います．

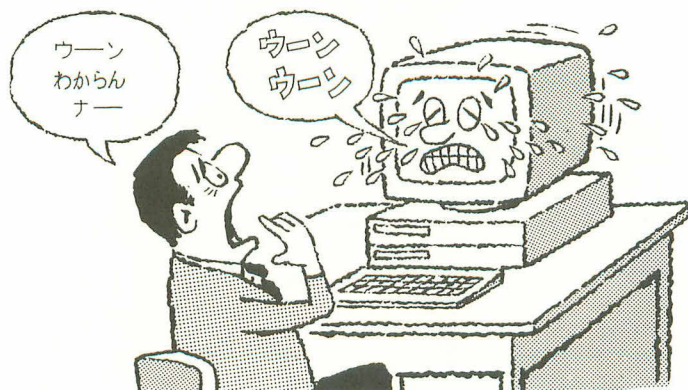
なお，この問題でいままであげたコーディング例は，理解しやすいように簡略化しありますが，実際にこのような問題をプログラミングするときは，いろいろな付随処理を考慮します．たとえば，各種エラーに対する処理などです．これらはすべて省略してありますのでご了解下さい．

8．コーディング結果をテストしよう

製品として世に出ていくものは，製品仕様として定められた機能を十分発揮できるかどうかを確かめるため，製造後に必ずテストされます．

プログラムとてその例外ではありません．むしろ，プログラムほど作成後のテストに力を入れる製品は他にないかもしれません．

ある調査では，ソフトウェア開発において，テストが全体作業の約半分を占めていることを示しています．この調査値は日本の企業663社のソフトウェア開発における実績値を平均したものです．みなさん自身，プログラムを作成するとき，作成中はできるだけ早く完成させるために，我を忘れて作業に熱中してしまうものと思いますが，後で振返ってみれば，その作業のなかでテストとエラー修正に費した時間が多いのに気付かれるのではないでしょうか(図8.1)．

図8.1　テストは楽なしごとではない

プログラムのテストにこれだけ力を注いでいる割には，プログラムの品質は他の製品に較べ，決して良いとは言えません．プログラムを稼動させた後も，しばらくの間は，思わぬエラーに悩まされるのが常です．

それだけに，プログラムのテストは慎重に，かつ効率的に行っていかなければなりません．特に，プログラムをモジュール分割した場合は，個々のモジュールのテスト順序がテストの効率とプログラムの品質に大きな影響をあたえます．

この章では，この問題に焦点をあて，効率的なテスト方法について検討してみることにします．

8.1 トップ・ダウン方式によるテスト方法を理解しよう

複数個のモジュールに分割したプログラムを効率的にテストするためには，トップ・ダウン方式によるテストがよいとされています．

そこでまず，トップ・ダウン方式によるテストとはどんな方法かを説明することにしましょう(図8.2)．

理解を容易にするために，簡単な例をもとに考えます．図8.3は，プログラムのモジュール構造の例を示しています．この例では，Aが最上位モジュールであり，直接従属モジュールB，C，Dを有しています．また，モジュールBはE，F，モジュールDはG，

図8.2 トップ・ダウン方式

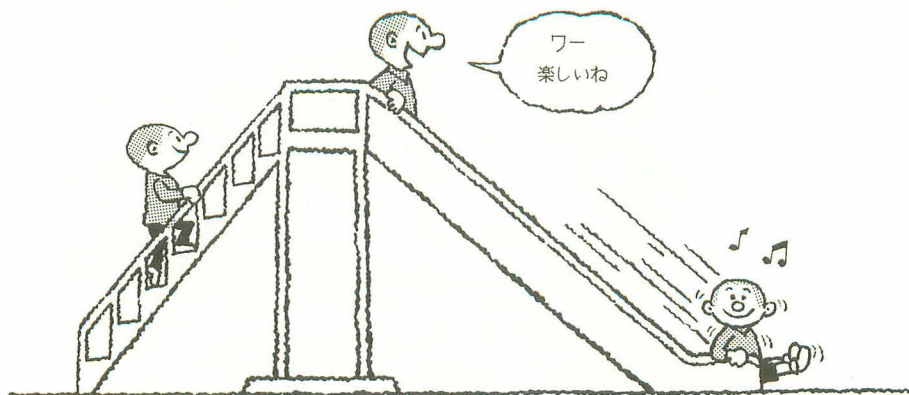
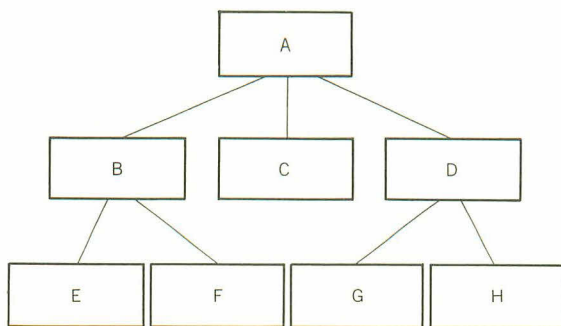



図8.3 モジュール構造の例

Hの直接従属モジュールを有しています．

トップ・ダウン方式によるテストでは，まず，モジュールAのテストから開始します．Aは直接従属モジュールB，C，Dを有していますから，Aをコーディングするさい，これら3つのモジュールがAのなかで呼び出されるはずです．たとえば，つぎのような形式になるでしょう．

```
モジュールA
main()
  double x,y,z;
      ⋮
  modB(x);
      ⋮
  modC(y);
      ⋮
  modD(z);
      ⋮
```

したがって，モジュールAをテストするためには，B，C，Dのモジュールが必要になりますが，この場合は，これら3つのモジュールは**スタブ**として扱います．スタブというのは，1種のダミー・モジュールのことです．

この段階では，Aをテストすることが主眼であり，B，C，Dも同時にテストするねら

いはありません．Aが正しく作動するかどうかがわかればよいのです．したがって，B, C, DはAのテストに支障がない程度のごく簡単なものにしておきます．たとえば，つぎのような形のものをとりあえず用意します．これらは上位モジュールから呼び出され，所定の引数のうけわたしが行われたことを確認するメッセージを打出すだけのものです．

```
モジュールB(スタブ)
modB(x)
  double x;
  {
    printf("modB CALLED\n");
  }

モジュールC(スタブ)
modC(y)
  double y;
  {
    printf"modC CALLED\n");
  }

モジュールD(スタブ)
modD(z)
  {
    printf("modD  CALLED\n")；
  }
```

このように，上位モジュールのテストのために仮にコーディングされたものがスタブなのです．B, C, Dをスタブでなく，実際の論理にそってコーディングしたものにすれば，A, B, C, Dが同時にテストされることになります．

もちろん，それでもよいのですが，A, B, C, Dを同時にテストして，うまく動かなかったとき，その原因をつきとめるとき，A, B, C, Dすべてのモジュールについて考えなければならなくなるでしょう．

これでは，同時に考えなければならない要因が多くなり混乱のもとになります．もともとモジュール化のねらいは同時に考えなければならない要因の数を少なくすることにあったわけですから，テスト段階でもその精神はいかしていく必要があります．

Aをテストするときには，Aだけに注意を集中し，他のモジュールのことはできるだけ考えないですむようにすることです．これがB，C，Dをスタブとして扱う主な理由です．

B，C，D，をスタブとして扱えば，B，C，Dの実際のコーディングができていない早い時点でもAのテストが可能になるという利点もあります．

もちろん，これらのスタブは，いずれそのモジュール本来の機能を実行するようにコーディングしなおされることになります．そのさいに，なるべく変更を少なくするためにも，スタブはできるだけ簡単にしておく方がよいのです．

このようにして，AのコードとB，C，Dのスタブを用いてAのテストを実施します．この時点でのテストは，下位モジュールがスタブとして扱われる関係上，ごく限られた一部の機能のテストだけにとどまることになりますが，基本的な実行順序は確立され，その妥当性が検査されることになります．

モジュールAのテストが一応終了したら，次はモジュールBのテストを行うことになります．この時点で，Bのスタブは本来のコードに拡張されます．そして，Bが呼び出すE，F用のスタブを作成します．

Bのテストには，Bのコードとともに，テスト済のAのコードも使用します．このことは，Bのテストと同時に，Aの再テストをも意味します．上位モジュールは早い段階でテストされ，その後，下位モジュールのテスト時に何度も何度も繰返しテストが行われ，結果として，品質の向上がはかれることになるのです．これがトップ・ダウン方式によるテストの大きな利点です．

図8.4にトップ・ダウン方式でのモジュールのテスト順序の全体像を示してあります．上位レベルから下位レベルへ，同じレベルでは左から右への順序で1つずつモジュールのテストを行っていくのが原則です．

実モジュールとスタブとの関係，スタブが実モジュールへ拡張されていく様子を図8.4から正しく読取って下さい．

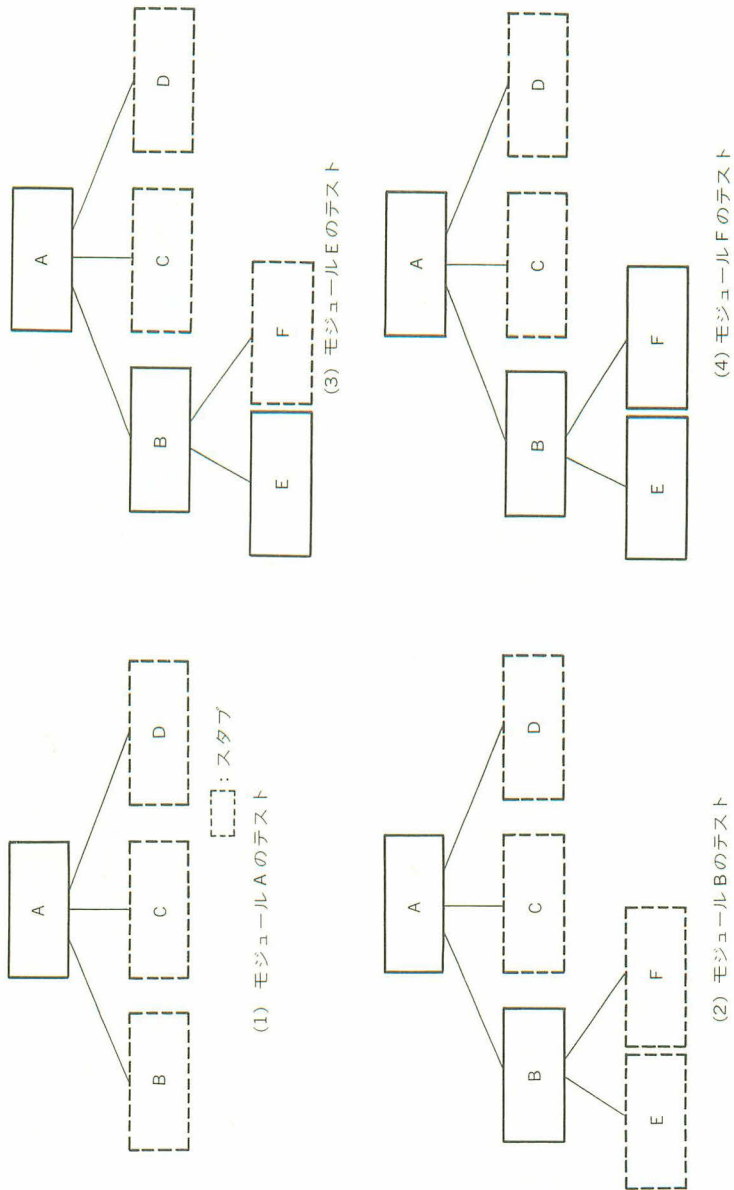
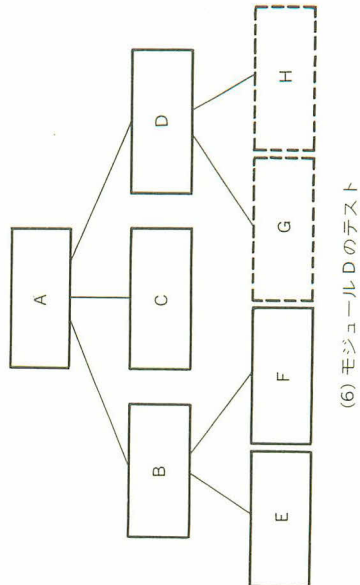
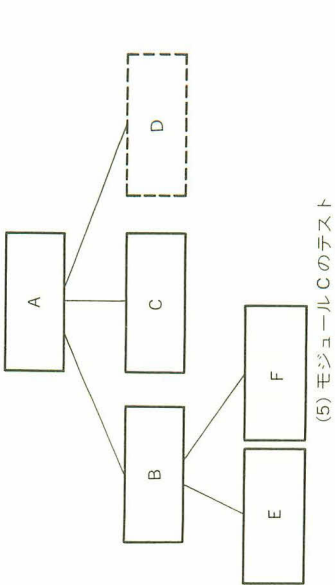
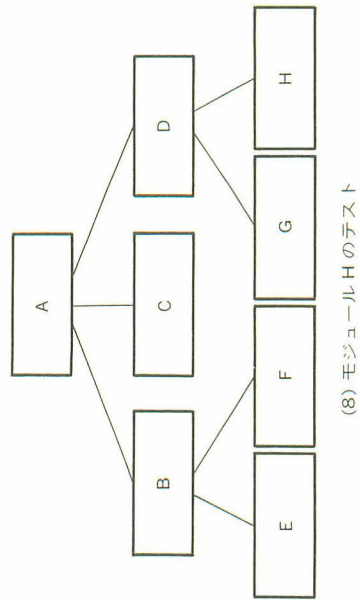
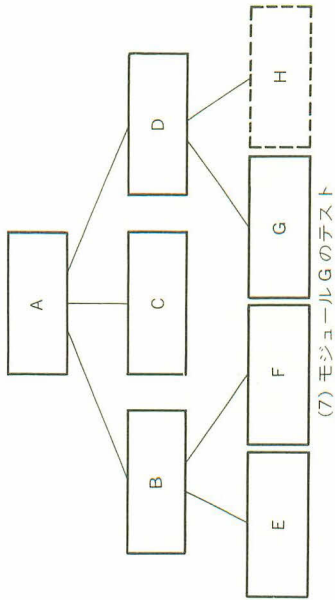
もちろん，図8.4に示した順序は原則であり，実際のプログラムをテストしていくときは，その内容やテスト・ケース等のかねあいでいろいろな変化がでてきます．

図8.4　トップダウン方式によるモジュールのテスト

(5) モジュールCのテスト

(6) モジュールDのテスト

(7) モジュールGのテスト

(8) モジュールHのテスト

一般的には，テスト順序はつぎのように考えるのが普通です．

(1) まず，正規処理のテストを行う
(2) 独立したエラー処理のテストを行う
(3) いくつかの条件が重なった時に予想されるエラー処理のテストを行う
(4) 限界処理(たとえば，テーブル・オーバフローなど)のテストを行う

トップ・ダウン・テストの思想をいかしながら，この順序でテストを行うようにすれば，それぞれのプログラムでのモジュールのテスト順序が最終的に決まることになります．次節で，これまでにとりあげた実例をもとにこのことを考えてみることにします．

最後に，トップ・ダウン方式によるテストの特徴についてまとめておきましょう．

●階層構造の最上位モジュール(主モジュール)を最初にテストできる(図8.5)．

最上位モジュールは，一般に，プログラム全体の制御を行い，最も重要なモジュールです．最も重要な部分を一番はじめにテストできるのは大きな利点です．重要な部分を最後にテストし，そこで欠陥がみつかれば，それまでにテストしたものがすべて無駄になってしまう可能性があります．

●基本的には，モジュール単位でテストすることになるので，エラーが発生したとき，あやまり部分をみつけやすい．

それまでうまく稼動していたものが，新たなモジュールを追加してテストした結果，うまく稼動しなくなったときは，新たに追加したモジュールに欠陥がある可能性が高く，そのモジュールを集中的に調べることにより，デバッグ作業を効率化できます(図8.6)

●下位モジュールのテストは，結果として，テスト済の上位モジュールの再テストを行っていることにもなる(図8.7)．

一般に，上位モジュールほど，プログラムでは重要な役割をはたしますので，重要な部分を何度も再テストできるのはプログラムの品質向上に大変役立ちます．

●基本的には，プログラムの実行順序にそってテストできる．

このことは，本番稼動の実環境のもとですべてのモジュールのテストができることを意味します．特に，モジュール間のインターフェースのテストが実環境のもとでできることの意義を大きく評価する必要があります(図8.8)．

●上位モジュールのテストでは，下位モジュールをスタブとして扱うため100％完全にテストすることができない．

図8.5 最上位モジュールを最初にテストできる

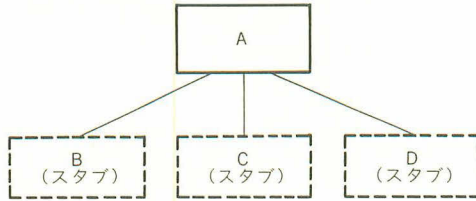



図8.6 新モジュールCのテスト時のエラーは新モジュールに原因のある可能性が高い

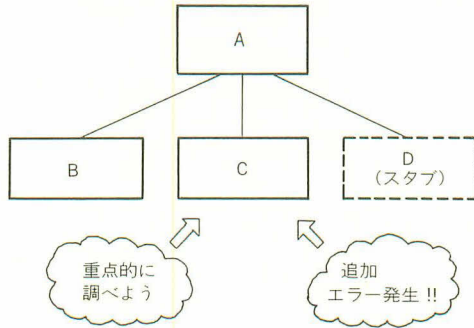



図8.7 モジュールBのテストはモジュールAの再テストにもなる

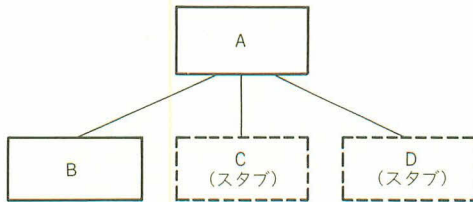
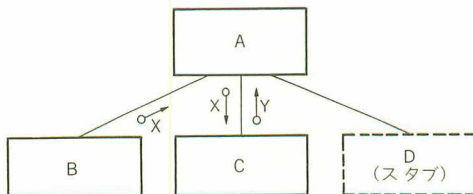



図8.8 実行順でテストすれば，実環境のもとでインターフェースのテストができる

100％完全なテストが可能になるのはすべての下位モジュールがスタブでなく実モジュールに拡張されたときです．したがって，それぞれの時点で，各モジュールがどこまでテストされたのかを注意深く管理する必要があります．

8.2　具体例でトップ・ダウン・テストの方法を考えてみよう

前節でトップ・ダウン・テストの原理については理解できたと思いますので，ここで例題にそれを適用してみましょう．

まず，いままでに何度もみてきた大気汚染度を測定する問題について適用してみます．この問題の最上位モジュールは「大気汚染度を測定する」です．最初にこのモジュールをテストします．その場合，3つの直接従属モジュールはスタブとして扱います(図8.9)．

コーディングは次のような形になっているでしょう．

```
大気汚染度を測定する(テスト・モジュール)
#define datasize 60
#define endHour 24
main()
{
  long ppm [datasize] ;
  int hour;
  double mean;
  for(hour=1;hour<=endHour;hour++)
    {
      getdata(ppm);
      cmean(datasize,ppm);
      putmean(hour,mean);
    }
}
```

1時間の測定値を得る(スタブ)

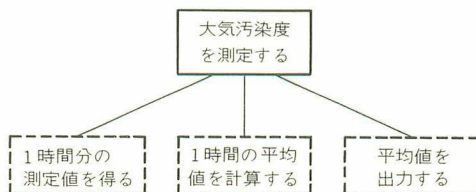
```
getdata(ppm)
  long ppm [ ]
  {
    printf("getdata CALLED\n");
  }
```

1時間の平均値を得る(スタブ)

```
cmean(datasize,ppm)
int datasize;
long ppm [ ] ;
{
  printf("cmean CALLED\n");
}
```

平均値を印刷する(スタブ)

```
putmean(hour,mean)
int hour;
double mean;
{
  printf("putmean CALLED\n");
}
```

図8.9　最上位モジュールのテスト

この段階では，テストの目的は最上位モジュール「大気汚染度を測定する」の制御論理をテストすることにあります．すなわち，1時間ごとに3つの従属モジュール getdata, cmean, putmean が呼出されるかをテストすることにあります．getdata, cmean, putmean の3つの関数そのものが正しく実行されるかどうかはテストの対象外です．したがって，これらはそれぞれスタブとして扱い，実際の論理でなく，呼出されたことを確認するだけのメッセージを書出すようになっています．

特に，cmean は実際には平均値を返す機能を実行するのですが，この段階ではそれを行っていませんので，main モジュールの命令の一部が実際とはちがったものになっています．main が実際に実行するときは

```
mean=cmean(datasize, ppm);
```

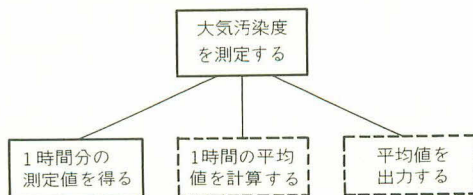
が実行されるのですが，この段階では

```
cmean(datasize, ppm);
```

になっています．

このように，テストのためにやむをえず実際のコーディングと異なった形にすることもありますが，当然スタブが実コードに拡張されたとき，main の方も実際のコードに変えます．しかし，このような変更はできるだけ少なくするように工夫することが必要です．

main モジュールのテストが終了すれば，次は「1時間の測定値を得る」モジュールをテストします．このモジュールのスタブを実コードにおきかえます(図8.10)．

図8.10 「1時間分の測定値を得る」モジュールのテスト

実コードはつぎのようになります.

1時間の測定値を得る(実コード)

```
#define endMin 60
getdata(ppm)
long ppm [ ] ;
{
  int minute;
  for(minute=0;minute<endMin;minute++)
    {
      scanf("%d",ppm [minute] );
      printf("%ld/n",ppm [minute] );
    }
}
```

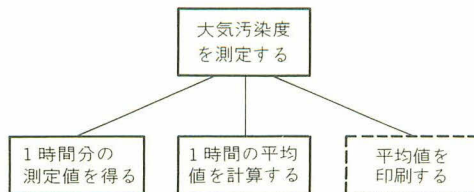
これにより, 1分ごとの測定値が60個, すなわち1時間分の測定値が ppm [0] から ppm [59] に入力される過程がテストされます. もちろん, この場合, テスト・データとして測定値を用意しておく必要があります.

このテスト用モジュールには, ppm [0] から ppm [59] に測定値が入力されたことを目でたしかめるために

printf("%1 da/n", ppm [minute]) ;

が換入してあります. テストが完了すれば, この命令を取除くことになります.

getdata モジュールのテストが終了すれば, つぎは mean モジュールのテストです(図

図8.11 「1時間の平均値を計算する」モジュールのテスト

8.11)．cmean モジュールはスタブから次のような実コードにおきかえられます．

```
平均値を計算する(実コード)
cmean(datasize,ppm)
int datasize;
long ppm [ ] ;
{
  double sum;
  int       i;
  sum=0.0;
  for(i=0;i<datasize;i++)
    {
      if(ppm [i] ==0)
       {
        sum=0.0;
        break;
       }
      else
          sum+=ppm [i] ;
    }
  return(sum/datasize);
}
```

このテストに際し，main モジュール内の

```
cmean(datasize, ppm) ;
```

の部分は

```
mean＝cmean(datasize, ppm);
```

に戻す必要があります．また，平均値が正しく計算されたことを確かめるために上の命令文に続いて

```
printf("%6.0 fln", mean);
```

を換入し，計算結果を目に見える形で表示してもよいですが，この例の場合は putmean がその役割をはたすように定められているので，mean のテスト時に putmean のテストも同時に行ってしまうのがよいでしょう．

なお，このモジュールのテストにさいし，テスト・データ(測定値)として，正しい値をもった測定値と装置が故障した時の測定値(0, 0)の両方を用意し，それぞれに対して適切な処理が行われることを確かめる必要があります．

9．新製品経済性評価プログラムの設計を行おう

これまでの説明でプログラム設計に対する基礎的な考え方は十分理解していただけたと思います．

そこで，この章ではその応用として，いままでとは違った少し本格的な問題に対するプログラム設計に取組んでみましょう．

9.1 問題仕様

《新製品経済性評価プログラム》

新製品を開発するにあたり，発売後の売行きとコスト(費用)を予測することにより，その製品が企業の利益に貢献するか否かを評価したい．そのためのプログラムを作成する．

発売後の売行きとコストは，企画部で設定した条件をもとに予測する．設定条件の

図9.1 新製品経済性評価報告書

＊＊新製品経済性評価報告書＊＊

年度	1月	2月	3月	4月	5月	6月	7月	8月	9月	10月	11月	12月	合計
1	−12,208	−11,395	−9,354	−8,385	−12,651	−10,533	−9,377	−7,067	−9,959	−8,554	−5,908	−4,225	−109,636
2	−7,216	−5,350	−2,192	17	−5,972	−3,167	−228	1,693	−2,126	14,892	34,369	38,700	−63,420
3	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
4	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
5	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

＊＊最終合計　25,685,133

詳細は後述する．

評価を行う期間は最長10年間とするが，実際の評価期間(年数)はプログラムの実行のつど，入力パラメータとして指定する．

指定された評価期間に対し，月毎に損益計算を行う．結果は図9.1のような形式で出力する．

毎月の損益計算は次のように行う

損益＝売上額－必要費用

売上額＝販売個数×販売単価

必要費用＝給与＋フロア経費＋諸経費
＋コミッション＋宣伝費
＋製造金額＋保管運送費
＋各月に配分された初期費

損益計算のために必要な各項目は，企画部が用意した予測のための基礎データにもとづいて算出される．

企画部の用意した基礎データは次のとおり．

P1　発売時この製品を販売するためにP1人のセールスマンを持つ．セールスマンはこの製品を専用に販売するものとする．

P2　セールスマンは，毎月P2の比率で退職する．

P3　入社して最初の月は，セールスマンはP3個を販売出来る．

P4　2月目以降セールスマンは彼の前月の売上に比べてP4の比率で売上個数を伸ばしていく．

P5　セールスマンは1月に最大P5個しか販売出来ない．

P6　発売して最初の月は，会社全体でP6個の売上げがある．

P7　毎月会社全体の売上個数は前月に比べてP7の比率だけ増加していく．

P8　1月当り会社全体で最大P8個までは売上をのばせる．

P9　発売時，セールスマンの初任給はP9百円とする．

P10 4ヵ月毎にセールスマンの基本給はP10の比率で上昇する.

P11 4ヵ月毎にセールスマンの初任給はP11の比率で上る.

P12 セールスマンには毎月彼の販売成績のP12に等しいだけの金額がコミッションとして支払われる.

P13 セールスマン10人毎にP13人の管理者を必要とする. 管理者はこの製品のためにだけ存在するものとする.

P14 この製品を販売するために最低P14人の管理者は必要である.

P15 管理者1人についてP15人のスタッフを必要とする.

P16 発売時管理者の初任給はP16百円とする.

P17 4ヵ月毎に管理者の基本給はP17の比率で上昇する.

P18 4ヵ月毎に管理者の初任給はP18の比率で上昇する.

P19 毎月P19の割合の管理者が退職する.

P20 発売時スタッフの初任給は毎月P20百円とする.

P21 4ヵ月毎にスタッフの基本給はP21の割合で上昇する.

P22 4ヵ月毎にスタッフの初任給はP22の割合で上昇する.

P23 毎月スタッフのP23の割合の人が退職する.

P24 当初, 製品の製造価格は1個当りP24百円とする.

P25 当初, 製品の販売価格は1個当りP25百円とする.

P26 製造価格は4ヵ月毎にP26の割合だけ高くなる.

P27 事務所のフロア・スペースを拡張する度に, 販売価格はP27の割合で高くなる.

P28 当初のフロア・スペースは管理者1人当りP 28 m^2の25%増しの広さでスタートする.
このフロア・スペースが管理者1人当りP 28 m^2より狭くなると, その時の10%に当る広さを拡張するものとする.

P29 発売時フロア・スペース1 m^2当り毎月P29百円の経費が必要である.

P30 フロア・スペースの経費は4ヵ月毎にP30の割合で増加する.

P31 什器備品, 清掃費等の諸経費として, その月のフロア・スペース経費のP31に相当する額が必要である.

P32 最初の4ヵ月間, 毎月の宣伝広告費として, その月の全セールスマンの給与のP32の割合を計上する.

P33　5ヵ月目以降毎月の宣伝広告費として，過去4ヵ月間に支払ったコミッション合計のP33に相当する金額を計上する．

P34　製品の保管費，運送費としてその月の製造金額のP34に当るだけの経費が計上される．

P35　発売にあたっての初期費は，発売当時のセールスマン1人当りP35百円必要とする．この費用は，これから評価しようとする年数または4年間のどちらか短い方を，その期間毎月均等に配分されるものとする．

P36　評価年数はP36で指定される．最大10年間とする．

(注)　4ヵ月毎とは発売月から数えての意味である．

9.2　問題の分析

9.1で述べられた仕様をもとに，この問題の分析をしてみることにします．特に，企画部が用意した評価のための基礎データを慎重に分析する必要があります．

分析のねらいは，P1～P36の基礎データが毎月の損益計算にどのように関係しているかを知ることにあります．

●販売個数の算定

まず，売上額を計算するために，販売個数を知る必要があります．販売個数に関係する基礎データはP6，P7，P8です．

発売して最初の月の販売個数はP6個です．その後，月毎にP7の比率で前月より販売個数が増えていきます．ただし，最大販売個数はP8個までです．

発売月の販売個数＝P6

発売月からnヵ月後の販売個数＝$P6(1+P7)^n$

　　ただし　$P6(1+P7)^n \leqq P8$

●販売単価の算定

販売単価に関係する基礎データはP25，P27です．発売月の販売単価はP25です．しかし，事務所のフロア・スペースが拡張されるたびに，P27の割合で高くなります．

発売月の販売単価＝P25

フロア・スペースが拡張されたときの販売単価＝旧販売単価(1＋P27)

このことから，フロア・スペースがきまらないと，販売単価がきまらないことに注意して下さい．フロア・スペースは管理者の数に依存しています(P28)．

●売上額の算出

売上額は販売個数に販売単価を乗じて算出できます

売上額＝販売個数×販売単価

次に，必要費用の算出方法について分析します．必要費用は下記の8つの費用の合計として算出できます．

人件費(給与)

フロア経費

諸経費

コミッション

宣伝費

製造金額

保管運送費

各月に配分された初期費

●人件費(給与)の計算

この製品の販売には，セールスマン，管理者，スタッフの3つの職種が関係しています．したがって，人件費はこの3つの職種の人件費の合計として算出できます．

人件費＝セールスマンの人件費＋管理者の人件費＋スタッフの人件費

この式を計算するにはセールスマン，管理者，スタッフの人数と給与額を知る必要があります．

●セールスマンの数の算出

セールスマンの数は，基本的には，販売目標(販売個数)によって決まります．

販売目標を達成するのに必要な数のセールスマンがいないときは，不足分を新規採用する必要があります．

n ヵ月目の会社全体の販売目標(個数)は次のような式で計算できます．

$$n\,ヵ月目の販売目標＝発売月の売上個数(1+P7)^{n}$$

これに対し，n ヵ月目の会社全体のこの製品に対する販売能力は次のように算出できます．

$$n\,ヵ月目の販売能力＝\sum_{J=1}^{n}(J\,ヵ月目入社のセールスマン数\times J\,ヵ月目入社セールスマンの販売能力)$$

入社後のセールスマンは毎月一定の比率(P2)で退職していますので，セールスマン数を算出するときはこのことを考慮しておく必要があります．

また，セールスマンの販売能力も毎月 P4 の比率で向上することも考慮する必要があります．すなわち，セールスマンの販売能力は入社月ごとに異なるのです．

このことは，入社月ごとにセールスマン数と販売能力を計算しなければならないことを

図9.2　セールスマン・テーブル

セールスマン・テーブル

S(1)	創業月入社セールスマン
S(2)	2ヵ月目　〃
	⋮
S(120)	120ヵ月目入社　〃

＊各月毎に退職者数を算出し，人数を更新する

図9.3　販売能力テーブル

販売能力(個数)テーブル

A(1)	1ヵ月入社セールスマン能力
A(2)	2ヵ月目入社　〃
	⋮
A(120)	120ヵ月目入社　〃

＊各月毎に販売能力向上率(P4)で能力値を更新する

意味しています.

この観点から，上式をもう一度よく確認しておいて下さい．この式を計算可能にするために，図9.2，図9.3に示すようなテーブルが必要になるでしょう.

このテーブルを用いれば，n ヵ月目に在籍しているセールスマンの数は

$$\text{n ヵ月目に在籍しているセールスマン数} = \sum_{i=1}^{n} S(i)$$

であり，その月に採用が必要な数は

新規採用必要数＝(販売目標個数－販売能力個数)/入社時セールスマン販売能力(P3)

になります．そして，その月のセールスマン総数は

セールスマン総数＝在籍セールスマン数＋新規採用数

です.

●管理者数の算出

必要な管理者の数は，セールスマンの総数によって決まります．セールスマン 10 人ごとに必要な管理者数が P13 で設定されます．また，管理者の最低必要人数が P14 で設定されています.

したがって，必要な管理者数は次の式で算出できます.

管理者総数＝(セールスマン総数/10)P13

もし，この式で計算した管理者総数が P14 より少なかったときは，

管理者総数＝P14

になります.

管理者も，セールスマンと同様，毎月一定の比率(P19)で退職します．また，初任給や基本給は入社月によって異なってきますので，人件費の計算のためにも，セールスマンと同様の入社月ごとの管理者数を貯えておくテーブルが必要になります(図9.4).

図9.4　管理者テーブル

	管理者テーブル
M(1)	創業月入社管理者数
M(2)	2ヵ月目　〃
	⋮
M(120)	120ヵ月目　〃

＊各月ごとに退職者を算出し，人数を更新する

このテーブルを用いれば，nヵ月目に在籍している管理者数は

$$n ヵ月に在籍している管理者数 = \sum_{i=1}^{n} M(i)$$

で求められます．

したがって，もし保有数＜必要総数ならば，その差の人数をその月に新規採用することが必要になります．

新規採用が必要な管理者数＝必要管理者総数－在籍管理者数

●スタッフ数の算出

必要なスタッフの数は管理者の数によってきまります(P15)．すなわち，管理者1人に対しP15人のスタッフが必要です．

必要スタッフ総数＝管理者総数 P15

その月に在籍しているスタッフ数が上式で計算した必要スタッフ数に満たないときは，その不足分を採用する必要があります．在籍スタッフ数は，セールスマンや管理者と同様，つぎのようなテーブル(図9.5)を作成し，そこから算出します．テーブルは人件費の算出

図9.5 スタッフ・テーブル

スタッフ・テーブル

	スタッフ・テーブル
F(1)	創業月入社スタッフ数
F(2)	2ヵ月目 〃
	⋮
F(120)	120ヵ月目 〃

＊各月ごとに退職者を算出し，
人数を更新する

のために必要になります．

$$n\text{ヵ月目に在籍しているスタッフ数}=\sum_{i=1}^{n}F(i)$$

新規採用が必要なスタッフ数＝必要スタッフ総数－在籍スタッフ数

ただし，在籍スタッフ数＜必要スタッフ数のとき

●退職者数の算出

セールスマン，管理者，スタッフとも毎月ある一定の比率(P2，P19，P23)で退職していきます．

各職種ごとにその月に退職する人数はつぎの式で算出できます．

退職者総数＝在籍者総数×退職率(P2，P19，P23)

たとえば，nヵ月目のセールスマンの場合は

$$\text{セールスマン退職者総数}=\left(\sum_{i=1}^{n}S(i)\right)\times P2$$

となります，また，退職者は入社の古い者から順に退職していくものとします．

いま，退職者総数が15名と算出されたとすると，更新前のセールスマン・テーブルの状況が図9.6のようであれば，更新後は図9.7のようになります．すなわち，1ヵ月目入社の10名がまず全員退職し，残りの5名は2ヵ月目入社の者が退職することになります．

先の人数計算のために用いるのは，更新後のテーブルを用いることになります．

●人件費(給与)算出の詳細

先に人件費は，セールスマン，管理者，スタッフの人件費の合計として算出する必要があることを述べました．そして，職種ごとの人件費を算出するためには，それぞれの人数

図9.6 更新前のセールスマン・テーブル

	(更新前)
S(1)	10
S(2)	10
S(3)	10
	⋮

図9.7 更新後のセールスマン・テーブル

	(更新後)
S(1)	0
S(2)	5
S(3)	10
	⋮

図9.8 セールスマン基本給テーブル

	セールスマン基本給テーブル
SS(1)	1ヵ月目入社セールスマン基本給
SS(2)	2ヵ月目 〃
SS(3)	3ヵ月目 〃
	⋮
SS(120)	120ヵ月目 〃

図9.9 管理者基本給テーブル

	管理者基本給テーブル
MS(1)	1ヵ月目入社管理者基本給
MS(2)	2ヵ月目 〃
MS(3)	3ヵ月目 〃
	⋮
MS(120)	120ヵ月目 〃

図9.10 スタッフ基本給テーブル

	スタッフ基本給テーブル
FS(1)	1ヵ月目入社スタッフ基本給
FS(2)	2ヵ月目 〃
FS(3)	3ヵ月目 〃
	⋮
FS(120)	120ヵ月目 〃

を知っおく必要があり，その人数の算出方法についていままで説明してきました．

特に初任給や基本給は入社月ごとに異なりますので，入社月ごとの人数を明確にする必要があり，そのためのテーブルの説明も行いました．

正確な人件費を求めるためには職種ごと，入社月ごとの基本給を算出しておく必要があります．

初任給

発売月：初任給＝P9(セールスマン)，P16(管理者)，P20(スタッフ)

4ヵ月ごとに：初任給＝旧初任給(1＋上昇率)

上昇率：セールスマン(P11)
管理者(P18)
スタッフ(P22)

基本給

入社月：基本給＝初任給
4ヵ月ごとに：基本給＝旧基本給(1＋上昇率)

上昇率：セールスマン(P11)
管理者(P17)
スタッフ(P21)

上記の算式によって，職種ごと，入社月ごとの基本給を算出し，次のようなテーブルに貯えておくことが必要です(図9.8，図9.9，図9.10)．

結果として，人件費はつぎのようにして求めることができます．

nヵ月目の人件費＝セールスマン人件費＋管理者人件費＋スタッフ人件費

$$= \sum_{i=1}^{n} S(i) \times SS(i) + \sum_{i=1}^{n} M(i) \times MS(i) + \sum_{i=1}^{n} F(i) \times FS(i)$$

●フロア・スペースと維持費の算出

必要なフロア・スペースは，管理者数によって決まります．フロア経費および諸経費(什器備品，清掃費等)はフロア・スペースの大きさに比例して算出できます．

必要なフロア・スペース

発売月：フロア・スペース＝管理者総数×P28×1.25

その他の月：フロア・スペース＜その月の管理者総数×P28 のとき
フロア・スペース＝前月フロア・スペース×1.1

フロア・スペース≧その月の管理者総数×P28 のとき
フロア・スペース＝前月フロア・スペース

フロア経費＋諸経費

発売月：フロア経費/m^2＝P29

4ヵ月ごとに：フロア経費/m^2＝(旧フロア経費/m^2)×(1＋P30)

各月のフロア経費＝その月のフロア・スペース×(フロア経費/m^2)

各月の諸経費＝その月のフロア経費×P31

各月のフロア経費＋諸経費＝その月のフロア・スペース×(フロア経費/m^2)
×(1＋P31)

●コミッションおよび宣伝費の算出

コミッションは，各月ごとに売上額の一定比率(P12)で支払われます．

その月のコミッション＝その月の売上額×P12

毎月の宣伝広告費は，最初の4ヵ月間に関しては全セールスマンの給与の一定比率(P32)が計上されます．

5ヵ月目以降は，過去4ヵ月間に支払ったコミッション合計の一定比率(P33)が計上されます．

宣伝広告費

最初の4ヵ月間：宣伝広告費＝全セールスマン給与×P32

5ヵ月目以降：宣伝広告費＝過去4ヵ月間のコミッション合計×P33

●製造金額および保管運営費の算出

製品の製造金額は，当初，1個あたりP25です．4ヵ月ごとに一定比率(P26)で高くなります．

製造金額

発売月：1個あたりの製造金額＝P25

4ヵ月ごとに：1個あたりの製造金額＝旧製造金額(1＋P26)

各月の製造金額＝その月の販売個数×1個あたりの製造金額

各月の商品の保管・運送費は，その月の製造金額の一定比率(P34)で計上されます．

保管・運送費

各月の保管・運送費＝その月の製造金額×P34

●初期費の配分の算定

発売にあたっての初期費は，発売当時のセールスマンの数に依存します．セールスマン1人あたりP35が必要になります．

初期費＝発売時のセールスマンの数(P1)×P35

初期費の各月への配分は次のように行われます．

評価年数(P36)＜4年のとき

配分された初期費＝初期費/(P36×12)

評価年数≧4年のとき

配分された初期費＝初期費/48

9.3　構造化設計

さて，いままで説明した問題仕様とそれにもとずく分析結果をもとに，いよいよ構造化設計にとりかかることにします．

▶問題構造の大要の把握

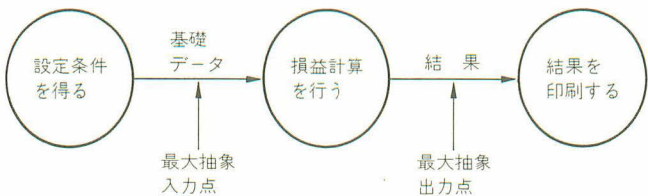
第6章で述べた構造化設計の手順にそって考えれば，まず第1に，問題構造の大要を把握することですが，この問題はその意味では大変明確です．

図9.11にこの問題の最上位構造を示します．この図から明らかなように，この問題は企画部が設定したデータをもとに，損益計算を行い，その結果を印刷出力することです．

したがって，問題の最初の分割には，源泉/変換/吸収分割技法を用いることができます．

図9.11の結果を用いて，問題の最上位構造を図9.12のように定めることができます．

この問題はここまではごく簡単なのですが，このあとが少し難しくなります．変換部分，すなわち，毎月の損益計算を行う機能をもっと詳細に分割していく必要があります．

図9.11 問題の大要

▶「損益計算」機能の分割

「損益計算」機能を部分機能に分割していくためには，そのなかでのデータの流れを正確に把握する必要があります．そのためには，9.2で分析結果をよく吟味しなければりません．

損益計算の基本は，収入と支出を正確に計算し，その差を求めることです．

損益＝収入－支出

毎月の収入を計算するためには，その月の販売個数と販売単価がわからなければなりません．先に分析したように，販売個数は設定データ(P 6，P 7，P 8)から簡単に求まりますが，販売単価をきめるにはフロア・スペースが影響してきます．したがって，フロア・スペースの算定をまたなければなりません．そして，フロア・スペースの算定のためには，管理者数が必要になります．管理者数をきめるためには，セールスマン数が必要です．セールスマン数をきめるのは販売個数です．この関係をわかりやすく図示すると図9.13のようになります．

図9.12 「経済性評価」プログラムの最上位構造図とインターフェース

経済性評価を行う
1 設定条件を得る
2 毎月の損益計算を行う
3 結果を印刷する

インターフェース

番号	入　力	出　力
1	—	設定データ(P1～P36)
2	設定データ(P1～P36)	毎月の損益
3	毎月の損益	—

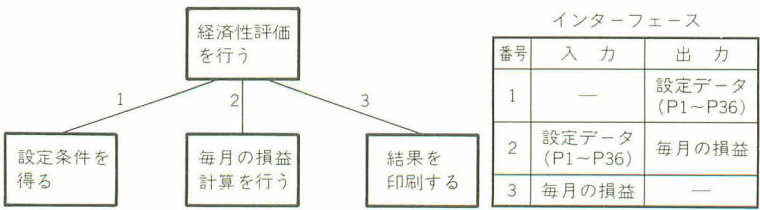
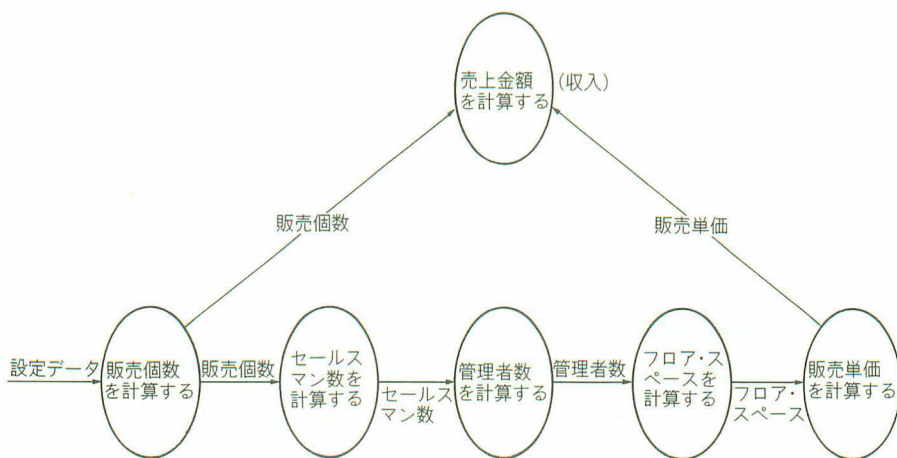



図9.13 「収入」計算のためのデータの流れと機能

次に支出の計算について考えてみましょう．支出項目は，先にみたように，給与，フロア経費，諸経費，コミッション，宣伝費，製造金額，保管運送費，各月に配分された初期費の8つです．

これら項目の計算方法については，すでに明らかにしましたが，計算のために必要なデータの流れを明確にすることによって，必要な機能も定義できます(機能は入力データから出力データへの変換過程であることを思い出して下さい)．

給与計算のためには，セールスマン数，管理者数，スタッフ数が必要になります(図

図9.14 「給与計算」のためのデータの流れと機能

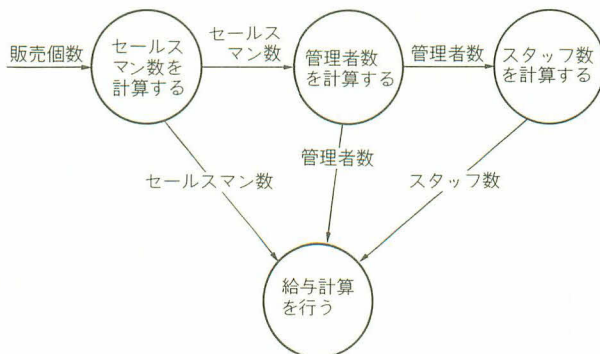



図9.15 「フロア経費」と「諸経費」計算のためのデータの流れと機能

9.14).

フロア経費，諸経費の計算のためには，管理者数が必要になります(図9.15).

コミッション計算には売上金額，宣伝広告費計算には全セールスマン給与とコミッション額が必要です(図9.16).

製造金額計算のためには販売個数，保管運送費計算のためには製造金額が必要です(図9.17).

特にふれませんでしたが，もちろん，各項目の計算にはこの他にいくつかの設定データが必要になります(モジュール・インターフェースを定義するときはこの設定データも必要になります).

以上のデータの流れを全体図として示すと図9.18のようになります.

図9.18は損益計算のための主要なデータの流れを示すと同時に，損益計算を行うために必要な機能も示ています.

図9.16 「コミッション」と「宣伝広告費」計算のためのデータの流れと機能

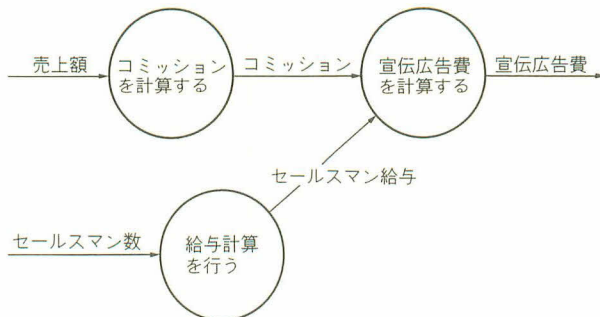



図9.17 「製造金額」と「保管運送費」計算のためのデータの流れと機能

▶全体構造の作成

これをもとに，新製品の経済性評価問題の機能展開図を作成すると図9.19のようになります．

この図で，販売個数計算，人数計算(セールスマン，管理者，スタッフ)，フロア・スペース計算の3つは販売活動をとらえるものとして求めている点に注意して下さい．

図9.18 「損益計算を行う」の全体データ・フロー・ダイヤグラム

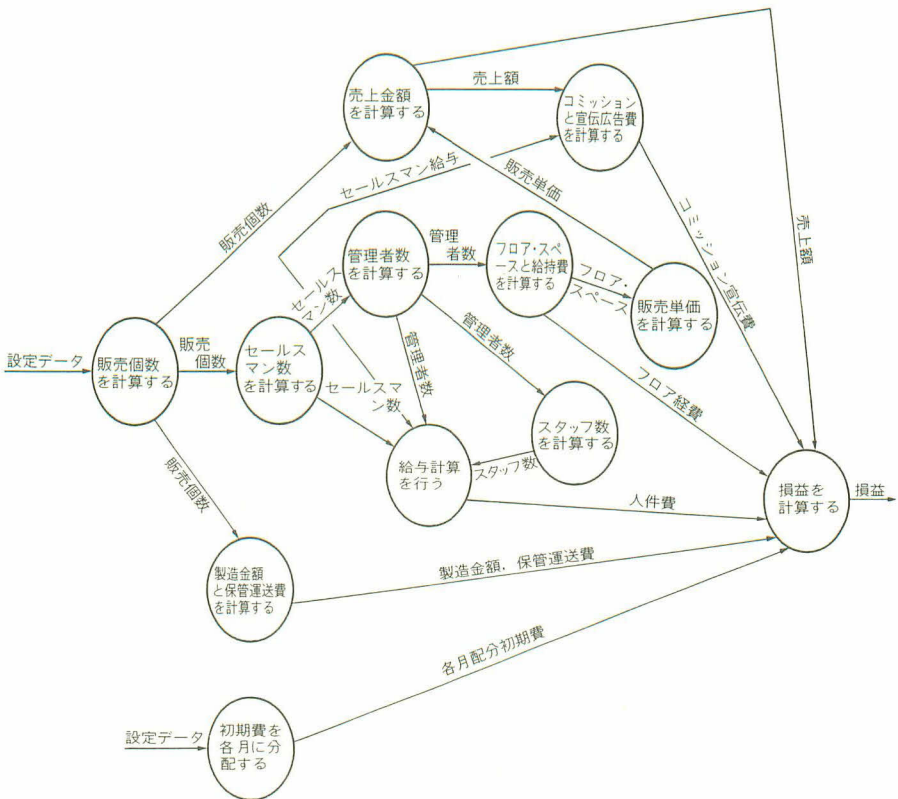
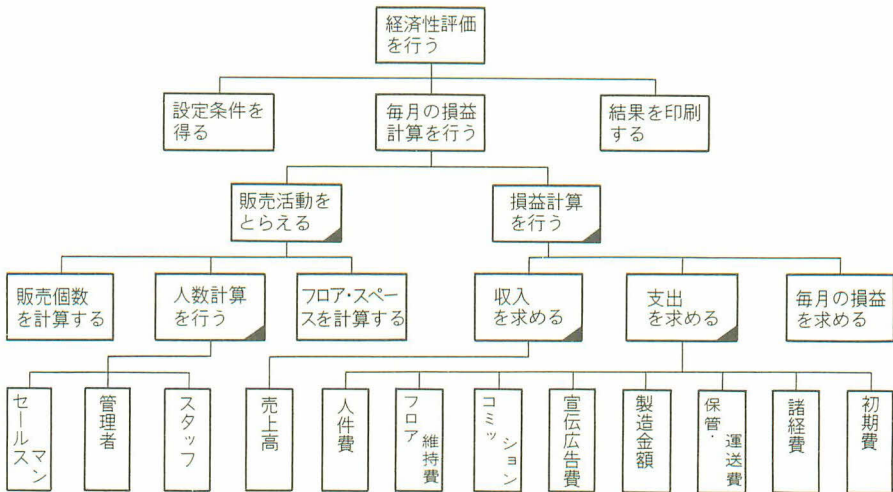



図9.19 「経済性評価」プログラム問題の機能展開図

ここでまとめた機能は，収入と支出を求めるために必要な販売活動をベースにした基礎データであり，直接の収入，支出計算と区分けすることにより，問題全体の理解度の向上に役立っています．

もし，これらの機能を収入，支出計算の一部に混在させれば，問題の焦点をぼかしてしまうきらいがあります．

なお，図9.19で機能を示す長方形枠の右下をぬりつぶしてあるものは，概念的な機能であり，問題の理解を容易にするために換入してあります．これらの機能によって，問題がより階層的に整理されていることをくみとっていただけたらと思います．

これらの機能は，実際にモジュール化するときは特に必要でなくなることが多く，省略されることがよくあります．この問題でも，コーディング対象になるモジュールを考える段階では，これらの概念的な機能は削除してあります．

▶モジュール構造の作成

図9.20はそのモジュール構造を示しています．また，図9.21はモジュール間インターフェースを示しています．図9.22はこれら記号で示されたインターフェースの意味をまとめたものです．

「退職人数を計算する」と「給与計算を行う」モジュールが共通モジュールとして設計

図9.20
「経済性評価」プログラムの
モジュール構造図

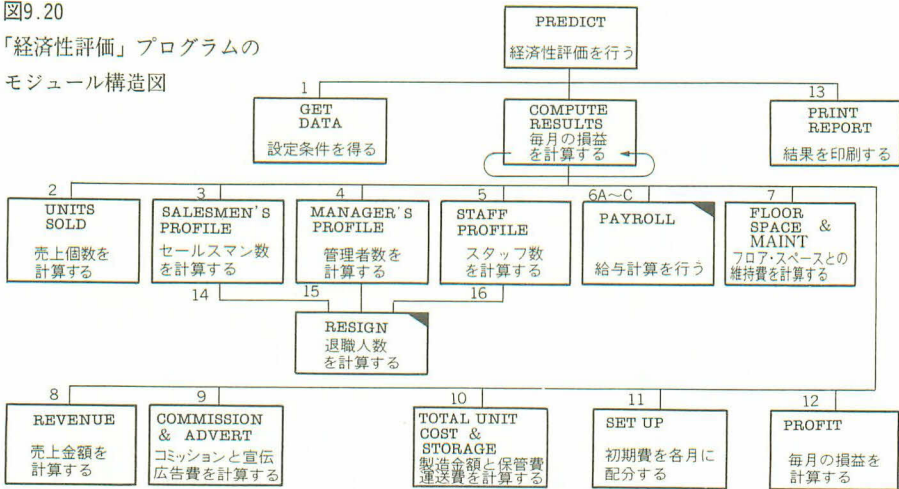



図9.21　「経済性評価」プログラムのモジュール・インターフェース

	入	出
1	無し	P1～P36
2	MONTH, P6, P7, P8	UNITS－SOLD
3	MONTH, P1, P2, P3, P4, P5, UNITS－SOLD	＊SALESMEN, ＊TOT－SALESMEN
4	MONTH, P13, P14, P19, TOT－SALESMEN	＊MANAGERS, ＊TOT－MANAGERS
5	MONTH, P15, P23, TOT－MANAGERS	＊STAFF, ＊TOT－STAFF
6A～6C	MONTH, START－SAL, CURRENT－RAISE, START－RAISE, PEOPLE	＊PEOPLES－SAL, ＊PEOPLES－BEGIN－SAL, TOT－PEOPLES－SAL
7	MONTH, TOT－MANAGERS, P28, P29, P30, P31	FLOOR－SPACE－AND－MAINT－COST, ＊FLOOR－AREA
8	MONTH, P25, P27, FLOOR－AREA, UNITS－SOLD	REVENUE
9	MONTH, P12, P32, P33, REVENUE, TOT－SALESMENS－SALS	COM－ADVERT－COST
10	MONTH, P24, P26, P34, UNIT－SOLD	TOT－UNIT－COST－AND－STORAGE
11	MONTH, P1, P35, P36	SET－UP－COST
12	MONTH, P25, P27, FLOOR－AREA, UNITS－SOLD	PROFIT－SAVE
13	P36, PROFIT－SAVE	無し
14～16	PEOPLE, TOT－PEOPLE, RESIG－FACTOR	PEOPLE, TOT－PEOPLE

(注)：(1) ＊印は入出力共用
　　(2) 6A～6C, 14～16はセールスマン，管理者，スタッフの3職種で共通

図9.22 「経済性評価」プログラムの主要変数説明図

変　　数	説　　　明
MONTH	発売以来の各月.
UNIT－SOLD	各月毎の販売個数.
SALESMEN (120)	発売以来, 各月毎に入社し, 在籍しているセールスマンの人数.
TOT－SALESMEN	全セールスマンの人数.
MANAGERS (120)	発売以来, 各月毎に入社し, 在籍している管理者の人数.
TOT－MANAGERS	全管理者の人数.
STAFF (120)	発売以来, 各月毎に入社し, 在籍しているスタッフの人数.
TOT－STAFF	全スタッフの人数.
START－SAL	発売時の初任給. ＝P9, P16 または P20
CURRENT－RAISE	基本給の上昇率. ＝P10, P17 または P21
START－RAISE	初任給の上昇率. ＝P11, P18 または P22
PEOPLE (120)	毎件計算の対象となる人. ＝SALESMAN, MANAGER, STAFF
PEOPLES－SAL (120)	各月毎に入社した人の基本給. 計算する時点で異なる. ＝SALESMENS－SAL (120), MANAGERS－SAL (120) または STAFF－SAL (120)
PEOPLES－BEGIN－SAL	各月に入社した人の初任給. 入社時点により異なる. ＝SALESMENS－BEGIN－SAL, MANAGERS－BEGIN－SAL または STAFF－BEGIN－SAL
TOT－PEOPLES－SALS	計算対象の職種全員の給与合計. ＝TOT－SALESMENS－SALS, TOT－MANAGERS－SALS または TOT－STAFF－SALS
FLOOR－SPACE－AND－MAINT－COST	事業所のフロア・スペースを確保し維持する費用
FLOOR－AREA	事業所のフロア・スペース面積
REVENUE	毎月の収入(売上金額)
COM－ADVERT－COST	毎月のコミッションと宣伝広告費合計.
TOT－UNIT－COST－AND－STORAGE	売上原価(製造金額と保管費, 運送費とを合計したもの)
SET－UP－COST	各月に配分される初期費.
PROFIT－SAVE (120)	各月毎の損益
RESIG－FACTOR	職種別退職比率 ＝P2, P19 または P23

(120)は配列として120個記憶域を確保する意味である.

されている点に注意して下さい. セールスマン, 管理者, スタッフの3つの職種に対するそれぞれの処理に対して呼び出されることになります. この部分のインターフェース・データの設定が工夫されている点をよく学んでおいて下さい.

また, フロア・スペースと諸経費の計算, コミッションと宣伝広告費の計算, 製造金額

と保管運送費の計算が，それぞれ１つのモジュールにまとめられている点にも気をつけて下さい．これらのモジュールは，強度的には連絡的強度になりますが，使用するデータの観点から密接に関係があり，一緒にした方がコーディングが効率的に行えることが予想されるのでまとめてあります．

その他のモジュールはすべて機能的強度になっており，また結合度としてはデータ結合になっていることをよく確かめて下さい．

なお，インターフェース・データの説明に用いた変数名は，この後のコーディングでそのまま用います．

9.4　モジュールの論理設計

構造化設計が終了すれば，次の作業は各モジュールの論理の設計です．

図9.20に示した各モジュールの論理を疑似コードで設計した例を図9.23〜図9.35に示してあります．

図9.23 「新製品の経済性評価を行う」の論理

```
初期設定を行う
入力データ(P1～P36)を読取る
入力データを書き出す
DOWHILE  評価期間(P36×12)の各月に対して
    売上個数の計算する
    セールスマン数を計算する
    管理者数を計算する
    スタッフ数を計算する
    給与計算を行う
    事業所の面積と維持費を計算する
    売上金額を計算する
    コミッションと宣伝広告費を計算する
    製造金額と保管費，運送費を計算する
    初期費を各月に配分する
    毎月の損益を計算する
    損益を損益テーブルに移す
ENDDO
各月ごとの損益と合計を書き出す
END  PREDICT
```

図9.24 「販売個数を計算する」の論理

```
IF 発売月
        売上個数＝P6
ELSE
        売上個数＝前月の売上個数(１＋P7)
ENDIF
IF 売上個数＞P8
        売上個数＝P8
ENDIF
END UNITS-SOLD
```

図9.25 「セールスマン数を計算する」の論理

```
IF 発売月
        初期設定を行う
ELSE
        退職者を算出して，在籍人数を更新する
        販売能力を更新する
        在籍人数による販売能力(個数)を計算する
        IF 販売能力＜売上目標
              必要数を採用する
        ENDIF
ENDIF
END SALESMENS-PROFILE
```

図9.26 「管理者の数を計算する」の論理

```
IF 発売月
        初期設定を行う
ELSE
        退職者を算出して，在籍人数を更新する
        必要な管理者数を計算する
        IF 在籍人数＜必要管理者数
              必要数を採用する
        ENDIF
ENDIF
END MANAGER-PROFILE
```

図9.27 「スタッフ数を計算する」の論理

```
IF 発売月
        初期設定を行う
ELSE
        退職者を算出して，在籍人数を更新する
        必要なスタッフ数を計算する
        IF 在籍人数＜必要スタッフ数
              必要数を採用する
        ENDIF
ENDIF
END  STAFF-PROFILE
```

図9.28 「退職者数を計算する」の論理

```
退職者総数を算出する
在籍者総数を更新する(新在籍者総数を求める)
在籍者テーブルを更新する
END  RESIGN
```

図9.29 「給与計算を行う」の論理

```
IF 発売月
        初任給＝発売時初任給(P9, P16, P20)
ELSE
        IF 昇給時(4ヵ月ごと)
              初任給＝旧初任給(1＋上昇率)
              DOWHILE  入社月＜その月
                基本給＝旧基本給(1＋上昇率)
              ENDDO
        ENDIF
ENDIF
DOWHILE  入社月≦その月
      入社月ごとの給与＝人数×基本給
      給与合計＝前月までの給与合計＋入社月ごとの給与
ENDDO
END  PAYROLL
```

図9.30 「フロア・スペースと維約費を計算する」の論理

```
IF 発売月
        フロア・スペース＝管理者総数×P28×1.25
        フロア経費＝P29
ELSE
        IF フロア・スペース＜管理者総数×P28
              フロア・スペース＝旧フロア・スペース×1.1
        ENDIF
        IF 4ヵ月ごと
              フロア経費＝旧フロア経費×(1＋P30)
        ENDIF
ENDIF
フロア・スペース経費と維持費の和＝フロア経費＋フロア・スペース(1＋P31)
END  FLOOR-SPACE-AND-MAINT
```

図9.31 「売上額を計算する」の論理

```
IF 発売月
        販売単価＝P25
ELSE
        IF 当月フロア・スペース＞前月フロア・スペース
              販売単価＝販売単価(1＋P27)
        ENDIF
ENDIF
売上額＝販売個数×販売単価
END  REVENUE
```

図9.32 「コミッションと宣伝広告費を計算する」の論理

```
コミッション＝売上額×P12
IF 最初の4ヵ月
        宣伝広告費＝全セールスマン給与×P32
ELSE
        宣伝広告費＝過去4ヵ月のコミッション合計×P33
ENDIF
コミッションと宣伝広告費の和＝コミッション＋宣伝広告費
END  COMMISION-AND-ADVERT
```

図9.33 「製造金額と保管運送費を計算する」の論理

```
IF 発売月
        製造単価＝P25
ELSE
        IF 4ヵ月ごと
              製造単価＝旧製造単価(1＋P26)
        ENDIF
ENDIF
製造金額と保管運送費の和＝製造単価×販売個数(1＋P34)
END  TOTAL-UNIT-COST-AND-STORAGE
```

図9.34 「初期費を各月に分配する」の論理

```
IF 評価年数(P36)＜4年
        配分した初期費＝発売時セールスマン数(P1)×P35/(P36×12)
ELSE
        配分した初期費＝発売時セールスマン数×P35/48
ENDIF
END  SET-UP
```

図9.35 「毎月の損益を計算する」の論理

```
損益＝売上額－(セールスマン給与合計
            ＋管理者給与合計
            ＋スタッフ給与合計
            ＋フロア経費と維持費の和
            ＋コミッションと宣伝広告費の和
            ＋製造金額と保管運送費の和
            ＋配分した初期費)
END  PROFIT
```

参考文献

1）G.J.Myers 著(國友義久，伊藤武夫 共訳)，『ソフトウェアの複合/構造化設計』，近代科学社，1979

2）國友義久 著，『効果的プログラム開発技法(第 3 版)』，近代科学社，1988

3）國友義久 著，『第 1 種情報処理技術者試験プログラム設計徹底マスター』，ソフトバンク，1991

4）G.J.Myers 著(久保未沙，國友義久 共訳)，『高信頼性ソフトウェア―複合設計』，近代科学社，1976

5）國友義久 著，『ストラクチャード・プログラミング入門(改訂第 2 版)』，オーム社，1988

6）E.Yourdon，L.L.Constantine 著(原田 実，久保未沙 共訳)，『ソフトウェアの構造化設計法』，日本コンピュータ協会，1986

7）J.Martin，C.McClure 著(國友義久，渡辺純一 共訳)，『ソフトウェア構造化技法』，近代科学社，1986

ディスク・サービスのお知らせ

本書の最後の章で設計した「新製品の経済性評価」プログラムのソースと実行形式オブジェクトをディスク頒布いたします．ソースはC++で書かれており，対応機種はPC-9801シリーズ，テストに使用したコンパイラはボーランドの「Turbo C++ ver 1.0」です．ご希望の方は下記の要領でお申し込みください．

なお，申し込みから発送までに2週間ほどかかることがございますのでご承知おきください．

▶プログラムのコンパイル，実行のしかたなどは，頒布ディスクのREADMEテキスト・ファイルに入っていますので，DOSのTYPEコマンドなどでごらんください．

ディスクの申し込み方法

申し込み用紙に住所/氏名を記入して，代金**3,000円(税，送料込み)**とともに現金書留でお送りください．

・メディア：5.25インチ2HDまたは3.5インチ2HD(いずれかを指定してください)

〈宛先〉

〒170 東京都豊島区巣鴨1-14-2

CQ出版㈱ C&E出版部

らくらく構造化設計係

■FINE SOFT　Cによるらくらく構造化設計

ディスク・サービス申し込み用紙

FS-CRR

送り先ご住所：〒

▶頒布希望メディア

(どちらかに印をしてください)

□ 5.25″ 2HD

□ 3.5″ 2HD

ふりがな

お名前：

TEL (　　)

◆著者略歴

國友義久（くにともよしひさ）

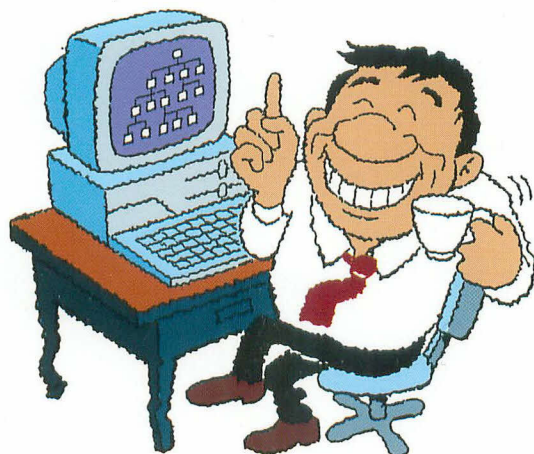
1961年	東京都立大学工学部電気工学科卒業 (株)リコーを経て
1964年	日本アイ・ビー・エム(株)入社 入社後，7年間システムズ・エンジニアとして，製造工業分野の客先をサポート．
1971年	同社教育センターに配属．主として，ソフトウェア・エンジニアリング，プロジェクト・マネジメント分野の教育を担当．
現在	研修センター所属．コンサルテーション・プログラム担当．
著書	『オンライン・ネットワークの構造的設計』（近代科学社），『効果的プログラム開発技法』（近代科学社）他

Cによる らくらく構造化設計

定価はカバーに表示してあります

1992年2月20日　初版発行

著者　國友　義久
発行人　神戸　一夫
発行所　CQ出版株式会社
〒170　東京都豊島区巣鴨1－14－2
☎03-5395-2122(出版部)
☎03-5395-2141(営業部)
振替　東京0-10665

写植　みやこワードシステム　印刷/製本　美和印刷株式会社
Printed in Japan

CQ出版社 定価2,400円(本体2,330円)

ISBN4-7898-3308-9 C3055 P2400E